RC24379 (W0710-066) October 11, 2007
Computer Science

IBM Research Report

On Approximating the Ideal Random Access Machine by Physical Machines

Gianfranco Bilardi
Dipartimento di Ingegneria dell'Informazione
Università di Padova
Italy

Kattamuri Ekanadham, Pratap Pattnaik
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
USA

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

On Approximating the Ideal Random Access Machine by Physical Machines

Gianfranco Bilardi[†] Kattamuri Ekanadham[‡] Pratap Pattnaik[‡]

Abstract

The capability of the *Random Access Machine* (RAM) to execute any instruction in constant time is not realizable, due to fundamental physical constraints on the minimum size of devices and on the maximum speed of signals. This work explores how well the ideal RAM performance can be approximated, for significant classes of computations, by machines whose building blocks have constant size and are connected at a constant distance.

A novel memory structure is proposed, which is *pipelined* (can accept a new request at each cycle) and *hierarchical*, exhibiting optimal latency $a(x) = O(x^{1/d})$ to address x, in d-dimensional realizations.

In spite of block-transfer or other memory-pipeline capabilities, a number of previous machine models do not achieve a full overlap of memory accesses. These are examples of machines with *explicit data movement*. It is shown that there are *direct-flow* computations (without branches and indirect accesses), which require time super-linear in number of instructions, on all such machines.

To circumvent the explicit-data-movement constraints, the *Speculative Prefetcher* (SP) and the *Speculative Prefetcher and Evaluator* (SPE) processors are developed. Both processors can execute any *direct-flow* program in linear time. The SPE also executes in linear time a class of loop programs that includes many significant algorithms. Even quicksort, a somewhat irregular, recursive algorithm admits a linear-time SPE implementation. A relation between instructions called *address dependence* is introduced, which limits memory-access overlap and can lead to superlinear time, as illustrated with the classical merging algorithm.

[†]Dipartimento di Ingegneria dell'Informazione Università di Padova, Italy. bilardi@dei.unipd.it
[‡]T.J. Watson Research Center, IBM, Yorktown Heights, NY, USA. {eknath,pratap}@us.ibm.com

1 Introduction

The *Random Access Machine* (RAM) [20] is an abstract model of computation, which has provided the basis for much design and analysis of sequential algorithms over the last several decades. The RAM can also be viewed as a rather idealized version of the *von Neumann architecture* [32], which has provided the basis for the design of many experimental and commercial computers for over half a century.

A central feature of the RAM is that any memory location can be accessed in just one basic step of the machine, no matter how large is the number of available memory locations. It is also customary to define the *execution time* of a RAM computation as the number of basic steps occurring in such a computation. The underlying assumption is that a basic step of the RAM can be carried out in an amount of time independent of the memory size, an assumption that appears impossible to meet in any physical machine. Specifically, the seeming impossibility arises from the conjunction of two principles: the principle of *maximum information density*, stating that a minimum volume of space is needed to store one bit of information, and the principle of *maximum information speed*, stating that there is a maximum velocity at which a bit of information can travel [16]. The first principle implies that, in a d-dimensional space, storing M bits requires a region whose linear dimension grows proportionally with $M^{1/d}$ which, by the second principle, translates into a similarly growing access time. In summary, one step of a RAM with M bits of memory can take, in the worst case, physical time $\Omega(M^{1/d})$. It is, on the other hand, rather straightforward to conceive of a memory organization where access time is bounded above by $O(M^{1/d})$, allowing a RAM step to be executed within the same time bound. Thus, a suitable d-dimensional physical machine can execute an arbitrary RAM computation of N steps in time $T = O(NM^{1/d})$. The broad question studied in this paper is whether this time bound can be improved, by more sophisticated machine organizations, for significant classes of RAM computations.

While our primary focus is on how fundamental physical constraints limit the performance of asymptotically large machines, the performance bottleneck arising from slow memory access relative to processor speeds has been a major concern in the design of actual machines. The issue has become particularly acute over the last decade or so, as conveyed by terms such as the memory wall, gap, or barrier [42, 40]. However, the issue had not escaped von Neumann's attention, even in the early stages of automatic computing.[§] Although, in current systems, the memory wall may be more closely related to present technological constraints such as I/O bandwidth limitations of silicon chips, than to speed of light or integration density, the structure of the problem faced by computer engineers is not essentially different from the structure of the problem that we are formulating in this paper. Correspondingly, approaches adopted in engineering practice can prove valuable in our context and vice versa.

There are two main avenues to mitigate the impact of memory latency on overall execution time. The first avenue consists in organizing the storage as a *hierarchy* of levels, from small and fast to large and slow, and in structuring the computation so that fast levels are accessed more frequently than slow levels. Computations that are so structured are commonly said to exhibit *temporal locality*. The second avenue consists in organizing the storage so as to enable *concurrency* of memory accesses, and in structuring the computation so as to take advantage of this concurrency. The two avenues can also interplay, as it happens in most computer systems where memory is partitioned

[§]See [32], Section 12.4: "... It shows in a most striking way where the real difficulty, the main bottleneck, of an automatic very high speed computing device lies: At the memory."

into sets of words with consecutive addresses, called *blocks* or *lines*, and transfers between levels of the hierarchy occur in blocks, concurrently for all the words of the block. This feature results in a performance advantage when computations tend to access consecutive addresses in near consecutive steps, a property referred to as *spatial locality*.

In this paper, we systematically explore the interplay between hierarchy and concurrency. Specifically, we show how to organize the storage as a *pipelined hierarchy*, where at each step a new read or write request can be submitted to memory, and is serviced with a latency corresponding to the level of memory involved in the request. Intuitively, such a powerful memory has a considerable performance potential, but it also calls for a processor capable of managing a large number of outstanding memory requests. Thus, we are led to study possible organizations of the processor side of the machine and, ultimately, to some interesting findings. Indeed, a third avenue emerges to mitigate the impact of memory latency, which could be called *memory bypass*: some of the information transfer between instructions, which in the RAM would take place via memory, is instead managed directly within the processor, with smaller latencies. Memory bypass can be viewed as a way to partially circumvent what Backus ([8], Sec. 3) calls the *von Neumann bottleneck*, that is the "tube" between the store and the central processing unit.

We underline that all our proposed organizations for both memory and processor are *scalable to arbitrary machine size*, under the fundamental physical constraints on minimum device size and maximum signal speed [16, 19]. This is established by exhibiting, for any given number of space dimensions d, designs that can be laid out in d-dimensional space so that the number of logic gates as well as the geometric distance to be traversed by each signal during one (logical) time step are independent of global machine size. Thus, physical time is actually proportional to the number of logical steps. Two-dimensional and three-dimensional layouts ($d = 2, 3$) are of particular interest in practice. One-dimensional layouts ($d = 1$) are also worth considering, as they capture the essence of several key issues in a simpler setting.

In the context of improving the performance of physical realizations of the RAM, we evaluate a specific machine $\mathcal{M}$ by establishing results of the following form: If P is a RAM program in class Π, which runs for N RAM steps on a given input, then P is executed by machine $\mathcal{M}$ in time $T \leq Ns(M)$. The factor $s(M)$ can be viewed as the *slowdown* of $\mathcal{M}$ with respect to the RAM or, in the parlance of computer architecture, as the *cycles-per-instruction (CPI)* metric for $\mathcal{M}$. Clearly, for a result to be of interest, $s(M)$ must grow more slowly than $M^{1/d}$, as the latter slowdown is rather straightforward to achieve, for any program that uses space at most M. Furthermore, the interest of the result increases with the extent and the relevance of the class of programs Π, although these aspects are not easily translated into a quantitative metric.

It is generally understood that the potential concurrency of a program is constrained by dependences between instructions. One key notion is that of *data dependence* [25], which holds between an instruction producing a value and another instruction using it. Within the course of this work, it has become apparent that a crucial distinction must be made between two cases of data dependence. In one case, which we call *functional dependence*, the output of the first instruction is used by the second one as an operand. In the other case, which we call *address dependence*, the output of the first instruction is used by the second one as the address of an operand. As we shall see, address dependences are a more serious obstacle to memory concurrency than functional dependences. An even greater obstacle arises from *control dependence*, whereby one instruction determines whether another one is executed or not. To allow for a gradual introduction of ideas, we will consider machines for the the execution of RAM programs with dependences of increasing

complexity. This approach will also help clarify which aspects of the memory and of the processor organization are needed to handle different types of dependences.

Next, we provide a roadmap of the paper. In Section 2, we review the definition of the Random Access Machine and introduce the key dependence relations among instructions. In Section 3, for any number of space dimensions $d \geq 1$, we develop pipelined, hierarchical memory organizations that can accept read or write requests at a constant rate and respond to a request involving address x in time $a(x) = O(x^{1/d})$. In Section 4, we develop a lower bound technique that applies to a wide class of machines, called machines with explicit data movement (EDM). In this class, data movement takes place only as explicitly specified by instructions. A consequence of this constraint is that any memory access latency can only be masked by overlap and cannot be eliminated altogether (as is done in bypass mechanisms). By making all data movements explicit, one can examine which of them can be overlapped and estimate time bounds more accurately. This class includes several previously proposed models of machines with hierarchical memory, in some case, as powerful as the organizations of Section 3. It is shown that there are direct-flow programs, *i.e.,* programs with no indirect addresses and no branches, any implementation of which will run in time superlinear in the number of instructions on any EDM machine. This analysis indicates, among other things, that a powerful memory by itself is not sufficient to achieve optimal performance. This motivates an investigation of processor organizations that engage in implicit data movements and can efficiently take advantage of the memory pipeline.

The investigation of non-EDM processors is carried out in Section 5, where an organization is developed with two variants: the *Speculative Prefetcher* (SP) and the *Speculative Prefetcher and Evaluator* (SPE). Both processors can execute arbitrary RAM programs. In the special case of *direct-flow* programs, the execution time is proportional to the number of executed instructions, as for the ideal RAM and unlike the other models previously considered in the literature. Both the SP and the SPE include a number of features that have become common in current microprocessors and make use of various forms of speculation, on data prefetch, on instruction operands, and on branch targets. The contribution of this work lies, in part, in the way these known mechanisms are combined and in part in the methodology that allows asymptotic analysis of algorithms on the proposed machines, in spite of the rather complex processor and memory structure. In particular, Section 6 is devoted to some case studies of algorithmic performance. It is shown that a class of loop programs can be automatically restructured to run in time proportional to the number of instructions on the SPE. This class includes matrix addition and multiplication, FFT, bitonic merging and sorting, finite difference solutions of some PDEs, and digital filtering. Recursive algorithms can also be programmed efficiently for the SPE, as illustrated with an optimal implementation of quicksort. A careful implementation of the standard algorithm to merge two sorted sequences is also considered: due to address dependences, the running time is superlinear in the number of executed instructions.

Finally, in Section 7, we put the results of the paper in some perspective and outline some directions for further investigation of the questions studied here.

The developments of this paper naturally relate to a vast amount of previous work, on models of computation, on algorithms, and on computer architecture. A full review of such work is beyond the scope and limits of this paper; however, a number of directly relevant references are reported in the subsequent sections.

2 The Random Access Machine and Related Concepts

In this section we formulate the framework of our investigation by (i) defining the abstract machine model for which we seek physical implementations; (ii) introducing the concepts of dependence between dynamic instructions of a program that are crucial to performance optimization, both when designing machines and when designing algorithms; and (iii) stating rules for our machine design that guarantee arbitrary scalability under physical constraints.

2.1 The RAM Model

The Random Access Machine (RAM) is a well known model of sequential computing [20], inspired by the von Neumann architecture [32]. In this section, we introduce a variant suitable to our purposes. In the theory of computation, the RAM is typically defined as a machine with a countably infinite number of storage locations, each capable of holding an arbitrary integer number. Instead, we define a finite version, RAM(M, P), a machine equipped with M locations of data memory and P locations of program memory; memory locations can hold strings with a bounded number of bits, as discussed later in the section. Indeed, we are interested in studying the performance and the cost of physical implementations as a function of machine size, as characterized by M and P. Accounting separately for program and data memory exposes more clearly the origins of the complexity in the implementation.

Data locations have addresses $0, 1, \ldots, M-1$; the content of location x is denoted by $m[x]$. Similarly, program locations have addresses $0, 1, \ldots, P-1$; the content of location x is denoted by $p[x]$. In addition, there is a special location, called the *program counter* and denoted pc.

The contents of locations pc, $m[0], \ldots, m[M-1]$, and $p[0], \ldots, p[P-1]$ represent the *state* of RAM(M, P), viewed as a discrete-time dynamical system. The transition function is naturally interpreted as the execution of the "current instruction", defined by the contents of location $p[pc]$. The set of possible instructions and their meaning, *i.e.*, the type of state transition caused by their execution, is considered next.

An *instruction set* $\mathcal{IS}$ is a finite set $\{\mathcal{J}_1, \ldots, \mathcal{J}_s\}$ of objects called *instructions*, whose format and meaning is discussed next. Instructions refer to data using the operand format $(\hat{x},\ x)$, where $\hat{x} \in \{-1, 0, 1\}$ specifies the so called *addressing mode*. Based on the mode, the function $value(\hat{x}, x)$ defines the value of data as follows: $value(-1, x) = x$ (*literal mode*); $value(0, x) = m[x]$ (*direct mode*); $value(1, x) = m[m[x]]$ (*indirect mode*). When $\hat{x} \neq -1$, we will sometimes find it useful to denote the address of operand location as $\tilde{x}$, defined as $\tilde{x} = x$ for $\hat{x} = 0$ and $\tilde{x} = m[x]$ for $\hat{x} = 1$.

Instructions are either of *data-processing* type or of *control-flow* type.

Data-processing instructions are of the form $\langle op,\ (\hat{x},\ x),\ (\hat{y},\ y),\ (\hat{z},\ z)\rangle$, where the instruction opcode op, denoting a binary operation, is followed by two source operand specifiers and one destination operand specifier (with $\hat{z} \neq -1$). The semantics of the instruction is to transform a current state, m_0, of the data memory into a next state, m_1, where $m_1[\tilde{z}] = value(\hat{x}, x)\ \ op\ \ value(\hat{y}, y)$. For all $w \neq \tilde{z}$, location $m[w]$ is unchanged, *i.e.*, $m_1[w] = m_0[w]$. After a data-processing instruction is executed, control flows to the next instruction in program memory ($pc_1 = pc_0 + 1$).

The main *control-flow* instruction is the *branch*, $\langle branch,\ (\hat{c}, condition),\ (\hat{t},\ target)\rangle$. When such an instruction is executed, if $value(\hat{c}, condition) \neq 0$, then control flows to the instruction at program memory position $pc_1 = value(\hat{t},\ target)$, else it flows to position $pc_1 = pc_0 + 1$. The memory remains unchanged. When $\hat{c} = -1$, the branch is unconditional and otherwise it is conditional. We shall also assume the existence of a *speculated branch* instruction to support static speculation.

The format is $\langle spbranch,\ (\hat{c}, condition),\ (\hat{t},\ target),\ (\hat{spt},\ sptarget)\rangle$, where *sptarget* specifies the speculated address of the next instruction to be executed after the branch. The value of the speculated target does not affect the semantics of a program, but can be helpful for performance. A *halt* instruction $\langle halt\rangle$ is also provided, which we assume to be always present in $p[0]$, and simply causes $pc_1 = 0$.

We also assume that, for any instruction, there is a corresponding *predicated* version, with an additional operand $(\hat{p}, predicate)$. The instruction is equivalent to its non-predicated version when $value(\hat{p}, predicate)$ is non-zero, and has no effect otherwise.

Finally, we introduce a performance-oriented *segment-size* instruction, $\langle ss,\ (\hat{s}, segmentsize)\rangle$, which does not alter the semantics of the program. Informally, it is meant to encode an indication of the number of instructions that could profitably be executed in an overlapped manner. The motivation for this instruction will be more clear in Section 6.

Remark. Technically, since a data-memory location must be capable of holding the address of an arbitrary data-memory location, it must be capable of storing at least $\log_2 M$ bits. Similarly, a program-memory location must be able to hold an instruction. A branch instruction can have a maximum of one data memory address and two program memory addresses. A non-branch instruction can have a maximum of 4 operands. Thus, each location in the program memory requires at least $max(4 * \log_2 M, \log_2 M + 2 * \log_2 P)$ bits. However, in most of our analysis, we will ignore the dependence of location size upon M and P, essentially considering it as a constant. This greatly simplifies many details, without compromising the main thrust of our considerations.

2.2 Instruction-Stream Dependences

A *program* $\mathcal{P}$ is a sequence of instructions, *i.e.,* $\mathcal{P} = (J_1, J_2, \ldots, J_q)$, where $J_i \in \mathcal{IS}$. In order to be executed, a program must be loaded into the program memory, with $p[i]$ containing the binary code for instruction J_i, for $i = 1, \ldots, q$.

It is generally understood that efficient execution of programs is constrained by various forms of dependences between their instructions. The dependence relations introduced below are similar in spirit to those that are standard in the field of compilers [25, 4], but they are also different in significant ways. Most notably, in the compiler context, relations are defined between static instructions in the program (the J_i's, in the above notation). In contrast, in our context, dependence relations are defined between instructions in the dynamic stream generated by executing the program on a given input.

When a program $\mathcal{P} = (J_1, J_2, \ldots, J_q)$ is executed for N steps, starting from an initial memory state m_0 and program counter $pc_0 = 1$, control will flow through a sequence of instructions, called the *instruction stream* $\mathcal{I} = \mathcal{I}(\mathcal{P}, m_o, N) = (I_1, I_2, \ldots, I_N)$, where $I_k = J_{pc_{k-1}}$. Correspondingly, the data memory will go through a sequence of states, $m_0, m_1, m_2, \ldots, m_N$, where m_k results from m_{k-1} by executing I_k.

We want to capture the situation in which an instruction I_j produces information used directly by a subsequent instruction I_k $(j < k)$. This happens when I_j produces the result v_j and stores it into location w, from which v_j is later read by I_k, with no intermediate instruction I_h $(j < h < k)$ writing into w. This situation is similar to data dependence in compiler theory. However, for our developments, it will prove crucial to further distinguish between the case where v_j is used by I_k as an operand value (functional dependence) and the case where it is used as an address (address dependence). For simplicity, in the next definition we assume that both I_j and I_k are

unpredicated data-processing instructions. The cases where I_k is a branch or predicated can be dealt with similarly.

Definition 1 *Let* $I_j = \langle op_j,\ (\hat{x_j},\ x_j),\ (\hat{y_j},\ y_j),\ (\hat{z_j},\ z_j)\rangle$ *and* $I_k = \langle op_k,\ (\hat{x_k},\ x_k),\ (\hat{y_k},\ y_k),\ (\hat{z_k},\ z_k)\rangle$ *be the* j*-th and* k*-th instructions in the instruction stream, respectively, and let* w *be a data-memory address.* ***FunctionalDependence****,* $j \xrightarrow{w} k$*, and* ***AddressDependence****,* $j \xrightarrow{*w} k$*, are defined as:*

$$
\begin{array}{lcl}
\mathit{produce(j,w,k)} & \equiv & (0 < j < k)\ \wedge\ (\tilde{z}_j = w)\ \wedge\ (\tilde{z}_h \neq w,\ \forall h:\ j < h < k) \\
\mathit{consumeValue(w,k)} & \equiv & w = \tilde{x}_k\ \vee\ w = \tilde{y}_k\ \vee\ (z_k = w \wedge \hat{z}_k = 1) \\
\mathit{consumeAddr(w,k)} & \equiv & (w = x_k \wedge \hat{x}_k = 1)\ \vee\ (w = y_k \wedge \hat{y}_k = 1) \\
j \xrightarrow{w} k & \equiv & \mathit{produce(j,w,k)}\ \wedge\ \mathit{consumeValue(w,k)} \\
j \xrightarrow{*w} k & \equiv & \mathit{produce(j,w,k)}\ \wedge\ \mathit{consumeAddr(w,k)}
\end{array}
$$

The dependence relation is conveniently extended to a fictitious instruction, I_0, to reflect that reading an unmodified location gets the value from the initial state of memory.

Definition 2 ***Functional Dependence****,* $0 \xrightarrow{w} k$*, and* ***Address Dependence****,* $0 \xrightarrow{*w} k$*, are defined as:*

$$
\begin{array}{lcl}
\mathit{produceInit(w,k)} & \equiv & (0 < k)\ \wedge\ (\tilde{z}_h \neq w, \forall\ 0 < h < k) \\
0 \xrightarrow{w} k & \equiv & \mathit{produceInit(w,k)}\ \wedge\ \mathit{consumeValue(w,k)} \\
0 \xrightarrow{*w} k & \equiv & \mathit{produceInit(w,k)}\ \wedge\ \mathit{consumeAddr(w,k)}
\end{array}
$$

Consider a chain of instructions where each instruction has a functional or an address dependence upon the previous one. It is intuitively clear that the indirect accesses generating the address dependences cannot easily[¶] be overlapped, since all the predecessors of an address-dependent instruction I must execute before the address needed by I can be even computed. These informal considerations motivate the definition of the address dependence depth of a program, which will play a significant role in subsequent sections.

Definition 3 *Consider an execution of program* $\mathcal{P}$ *on a particular input, resulting in the instruction stream* $\mathcal{I} = (I_1, \ldots, I_N)$*. A subsequence* $I_{j_0}, I_{j_1}, \ldots, I_{j_L}$*, with* $j_0 < j_1 < \ldots < j_L$*, forms a* ***dependence chain*** *if* I_{j_ℓ} *has a functional or an address dependence upon* $I_{j_{\ell-1}}$*, for* $\ell = 1, 2, \ldots, L$*. The* ***address dependence depth*** *(AD-depth) of a chain is the number of pairs of consecutive instructions in the chain related by address dependence. The address dependence depth* D *of an execution is the maximum AD-depth of any chain. The AD-depth* $D(K)$ *of* $\mathcal{P}$ *is the maximum AD-depth of any execution corresponding to an input of size* K*.*

The following types of program are often useful for conceptual developments.

Definition 4 *A program* $\mathcal{P}$ *is* ***straight-line*** *if all of its instructions are of the data-processing type, except for the last instruction which is a halt. A straight-line program using only literal or direct addressing is called a* ***direct-flow*** *program.*

To relate the preceding definitions, we may observe that a direct-flow program is characterized by lack of branches and AD-depth $D = 0$.

[¶]We cautiously use the qualifier "easily," because one cannot rule out a priori the possibility of predicting addresses and overlapping the corresponding (speculative) accesses.

2.3 Extensibility under Physical Constraints

We explore machine design and performance under very fundamental implementation constraints, likely to remain in force as technology evolves and larger systems are built. Thus, we focus on systems that are *extensible*, as specified by the following properties:

1. The system is formulated as an interconnection of modules of relatively simple functionality, specifying its mapping onto a physical system, including a placement in physical space of the modules and their interconnections.
2. The system design is parameterized with the size of the system, so that the same design can be instantiated in physical space for different sizes.
3. The execution time of a computation on the system is expressed in number of steps, as a function of the size, where each step is a state transition of the physical implementation.
4. The time taken for a state transition must remain invariant as the system grows in size.

Systems must be implemented ultimately in a d-dimensional space (typically with $d = 2, 3$), where physics sets lower bounds on device size and upper bounds on signal speed. (For models of computations incorporating such constraints see, *e.g.*, [19, 16]). Then, extensibility can be ensured by organizations formulated in terms of *cells* that can be realized with a constant amount of logic and can be embedded in a d-dimensional array with direct links between cells spanning at most a constant number of array edges. Our designs fall in this class.

3 Pipelined and Hierarchical Memory Organizations

In this section, we introduce novel memory organizations that, together with processor organizations developed in later sections, will lead to machines for efficient execution of RAM programs.

Traditionally, to reduce the memory bottleneck, designers have resorted to memory organizations that are *hierarchical* (to exploit the temporal locality of the computation), that support *block transfer* (to take advantage of the spatial locality of the computation), and that are *pipelinable* (to exploit concurrency of memory accesses). Memory hierarchies with block transfer capabilities have received a great deal of attention, from the perspective of architectures, compilers, algorithms, and libraries (see, *e.g.*, [21, 29, 2, 3, 34, 5, 41, 35, 38, 22, 31, 7, 15, 39]).

In contrast, relatively few studies have focussed on memory pipelining. Indeed, until recently, memories of most commercial machines were scarcely pipelinable, perhaps with the notable exception of vector processors. However, the number of simultaneous outstanding memory references has been increasing and is currently of the order of 10. Overlapping of memory references as a systematic approach to hide memory latency has been strongly advocated by Burton Smith, originally with the proposal of the HEP computer [36] and more recently in the context of the Tera computer [6]. Similar ideas have been later utilized in theoretical studies of shared memory simulations by fixed topology networks [37] and have subsequently inspired the realization of the SB-PRAM machine [1]. Speedups enabled by parallelism among operations are considered by [28], in uniprocessors with non-hierarchical pipelinable memories. Lower bound arguments developed in [16] show that, in parallel machines where performance is limited by speed of light (rather than by less fundamental overheads), concurrency of accesses cannot fully mask latency if locality of reference is not simultaneously exploited. Hardware and software *prefetching* techniques (see, *e.g.*, [30]) can be viewed as efforts to take advantage of concurrency among memory references, in a hierarchical context. The algorithmic advantages of pipelining the memory hierarchy have been

explored in [9], based on a model called the Pipelined Hierarchical RAM (PH-RAM).

The above cited studies and others provide strong motivations of both a practical and a theoretical nature to consider the pipelining of accesses in a hierarchical memory and to study the combined effects of concurrency and locality. The remainder of this section will focus on physical realizations of the following type of memory.

Definition 5 *A* ***pipelined hierarchical memory*** *satisfies the following properties.*

- *Memory consists of M locations with* ***addresses*** $0, 1, \ldots, M-1$. *The content of location x (at an implicitly understood time) is a word, denoted by $m[x]$.*
- *Memory has a single* ***input port*** *through which it accepts read and write requests and a constant number of* ***output ports*** *through which it delivers any responses. (Multiple output ports can simplify congestion management.)*
- *A* ***read request*** *is a packet of the form load$(x; tag)$; in response, the memory will deliver a packet at an output port, with the information $(x, m[x]; tag)$. The tag field, which is not modified by the memory, provides a convenient mechanism for the processor to associate with the request information that will be useful in processing the response.*
- *A* ***write request*** *is a packet of the form store(x, v) and will cause the content $m[x]$ of location x to be set to value v.*
- *Request packets (RPs) directed to the same memory location take effect in the* ***same order as they are submitted*** *to the input port.*
- *The input port and the output ports are placed within a constant distance of each other, at one corner of the memory layout.*
- *In each time step, the input port can accept one RP and each output port can deliver one RP (with the response to an earlier read request).*
- *Memory performance is characterized by two metrics. The* ***access function*** *$a(x)$ denotes (an upper bound to) the time that elapses between submission at the input port of a read RP for address x and delivery at an output port of the corresponding result. The* ***pipeline period*** *$p(M)$ denotes the minimum time between subsequent submissions of RPs, in a sustained fashion.*

Without loss of generality, we assume that $a(x+1) \geq a(x)$, *i.e.,* that the locations are numbered in order of non-decreasing access time. The restriction on the number of ports is in no way imposed by physical constraints and, as we will discuss, ought to be lifted in further investigations.

For a memory physically laid out in d dimensions, it is straightforward to see that constraints on minimum device size imply that location x must be at distance $\Omega(x^{1/d})$ from the memory ports. Then, the constraint on maximum signal speed implies that $a(x) = \Omega(x^{1/d})$. It is quite straightforward to devise a memory organization with $a(x) = O(x^{1/d})$, if distinct accesses are not overlapped. It is more challenging to attain the same bound if accesses are pipelined. In fact, responses to different addresses submitted at different times will generally compete for the bandwidth toward the output ports. The problem then becomes to control congestion in such a way that each individual memory request does not get delayed by more than a constant factor, due to routing conflicts with other requests.

The main contribution of this section is a novel pipelined, hierarchical memory organization scheme that can be specialized to yield optimal access time $a(x) = O(x^{1/d})$, with $p(M) = O(1)$, that is, requests are accepted at a constant rate. The result holds for any $d \geq 1$. Two-dimensional and three-dimensional layouts ($d = 2, 3$) are of particular interest in practice. One-dimensional

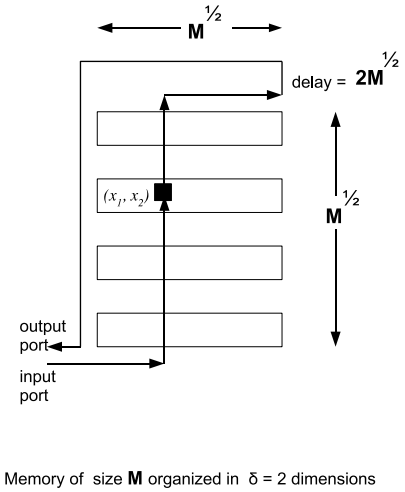


Figure 1: A non-hierarchical, pipelined memory organization.

layouts ($d = 1$) are also worth considering, as they capture the essence of several key issues in a simpler setting. Regardless of its potential significance, the case $d > 3$ is included in this study without effort, since our constructions are naturally uniform with respect to d.

As a building block for the pipelined hierarchy, Section 3.1 introduces a non-hierarchical, pipelined memory. This block is of independent interest for a number of results in this paper, which rely only on the pipelinability of the memory and not on its hierarchical nature.

3.1 A Pipelined Memory

We describe a δ-dimensional pipelined *non-hierarchical* memory of size M, for simplicity a power of δ, organized as a δ-dimensional array of memory cells, whose dimensions can be expressed as $[0..M^{1/\delta}-1, \ldots, 0..M^{1/\delta}-1]$. The input port is placed near the origin $[0, \ldots, 0]$. All requests enter at the origin, traverse a path that goes through the target cell, exit at the diagonally opposite end of the δ-dimensional array, and return to the origin via a path external to the array, as illustrated by Figure 1, for $\delta = 2$. Specifically, a read request for the cell $[x_1, x_2, \ldots, x_\delta]$, first travels for x_1 units along the first dimension, *i.e.*, $[0..x_1, 0, 0, \ldots, 0]$, then x_2 units along the second dimension, *i.e.*, $[x_1, 0..x_2, 0, 0, \ldots, 0]$, and so on until it reaches the target at $[x_1, x_2, \ldots, x_\delta]$. It collects the data from the cell and travels along $[x_1, x_2, \ldots, x_{\delta-1}, x_\delta + 1..M^{1/\delta} - 1]$ and then along $[x_1, x_2, \ldots, x_{\delta-1} + 1..M^{1/\delta} - 1, M^{1/\delta} - 1]$ and so on until the last node $[M^{1/\delta} - 1, \ldots, M^{1/\delta} - 1]$ is reached. The total path length from the origin to the diagonally opposite node is $\delta M^{1/\delta}$. Thus, including the return path, access to any location in this array takes time $\mu = 2\delta M^{1/\delta}$. A write request is handled similarly, with the path stopping at the target cell.

The proposed organization is highly pipelinable, as it can accept one request in each time step, *i.e.*, $p(M) = 1$. In fact, assuming each step in the traversal described above takes the same amount of time, no two requests collide at any point in their traversal. It is also clear that requests to a given cell arrive at the cell in the order they are submitted. The memory is not hierarchical as access to any location in the array takes the same amount of time, that is $a(x) = 2\delta M^{1/\delta}$, for $x = 0, \ldots, M - 1$. We summarize the result next.

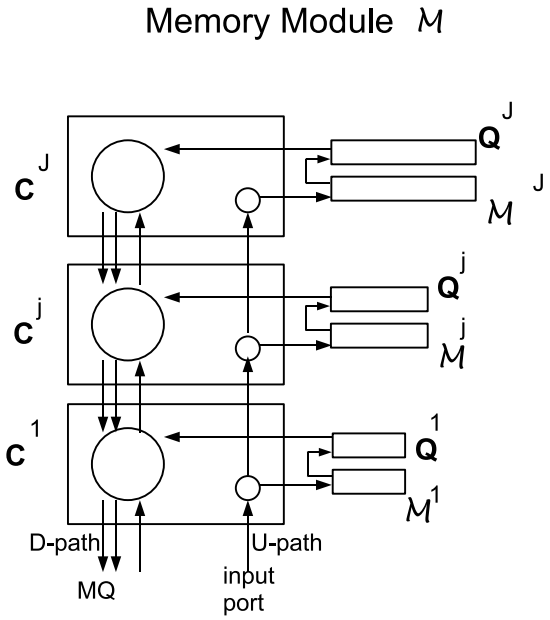


Figure 2: A pipelined hierarchical memory organization. Memory locations are organized into J levels, $\mathcal{M}_1, \ldots, \mathcal{M}_J$. Each $\mathcal{M}_j$ is a pipelined memory with M^j cells and μ^j response time. Responses from $\mathcal{M}^j$ are output into a FIFO queue Q^j. A backbone organized as a linear array of controllers $C^1, \ldots, C^J$ routes the requests to the appropriate memory levels (along the U-path) and the responses back to an output queue MQ (along the D-path).

Proposition 1 *For $\delta \geq 1$, the preceding construction yields a memory organization of M cells that can be laid out in δ dimensions and achieves period $p(M) = 1$ and access time $a(x) = O(\delta M^{1/\delta})$.*

3.2 Pipelined Hierarchical Memory: Overview

The memory $\mathcal{M}$ developed in this section consists of a sequence of *levels* $\mathcal{M}^1, \mathcal{M}^2, \ldots, \mathcal{M}^J$, interconnected by a *backbone* (see Figure 2). For each level, $1 \leq j \leq J$, $\mathcal{M}^j$ is a pipelined memory of M^j locations (j is a superscript, not an exponent), which can accept one request packet (RP) per time step and complete the memory operation after a fixed number, μ^j, of time steps. If the RP is for a read, then is loaded with the value of the target location and sent into an output queue Q^j. This is a FIFO queue which can accept one RP per step and output up to a fixed number of the enqueued RPs. A systolic implementation of the queue is feasible, in the spirit of [27].

The backbone, fully described in Section 3.3, consists of a bidirectional linear array of controllers C^j ($j = 1, \ldots, J$). An RP directed to a location x in level $\mathcal{M}^j$ is submitted to controller C^1, routed "upwards" on the U-path by the backbone to controller C^j, and submitted to $\mathcal{M}^j$. For a read, the RP with the memory response goes to Q^j, eventually enters the backbone at C^j, is routed "downwards" on the D-path to controller C^1, and is finally inserted into a queue, MQ, described in Section 3.5. Ultimately, when we specialize the organization for d-dimensional layouts, the $\mathcal{M}^j$'s will be organized as the $\delta = (d-1)$-dimensional arrays described in Section 3.1. However, for most of our developments, we will refer to arbitrary level parameters M^j and μ^j. Memory locations are

assigned consecutive addresses, starting at 0, in level order. Therefore, the level $\ell(x)$ containing address x is uniquely determined by the relation $M^1 + \ldots + M^{\ell(x)-1} \leq x < M^1 + \ldots + M^{\ell(x)}$. Let us now define $a(x)$, the *access time* to location x, as the worst-case duration of the interval from the time when an RP targeted to location x is submitted to $\mathcal{M}$ to the time when the response to that RP is inserted into MQ. Clearly, for the RP to travel up to level $\ell(x)$, to be serviced once at level $\ell(x)$, and to travel down to MQ it takes at least time $\ell(x)$, $\mu^{\ell(x)}$, and $\ell(x)$, respectively, so that $a(x) = \Omega(\ell(x) + \mu^{\ell(x)})$. Our objective is to achieve $a(x) = O(\ell(x) + \mu^{\ell(x)})$. The challenge lies in designing the downward routing algorithm of the backbone so that the time spent by an RP both in $Q^{\ell(x)}$ and in the backbone are suitably bounded. Routing on a linear array has been widely investigated; however, the version of the problem arising here differs from previously studied scenarios in two key respects: (i) the objective is to bound the routing time to within a constant factor of *source-destination distance* for individual messages (not to within a constant factor of the *network diameter* as, *e.g.*, in most studies of permutation routing) and (ii) submission of messages is continuous, although subject to some constraints (most routing analysis has been developed for batch mode).

The next subsection is devoted to a detailed description of the backbone. The subsequent subsection is devoted to performance analysis and establishes that $a(x) = O(\ell(x) + \mu^{\ell(x)})$. In d dimensions, level $\mathcal{M}^j$ can be realized as a $(d-1)$-dimensional pipelined memory of size $M^j = (j)^{d-1}$, with $\mu^j = O((d-1) * j)$, as in Proposition 1, with $\delta = d - 1$. One can easily see that $M^1 + M^2 + \ldots + M^{j-1} = \theta(j^d)$, whence $\ell(x) = O(x^{1/d})$. Therefore, optimal access time $a(x) = O(\ell(x) + \mu^{\ell(x)}) = O(dx^{1/d})$ is achieved.

3.3 Structure and Routing Algorithm of Backbone

The memory backbone is structured as a linear array of controllers $C^1, C^2, \ldots, C^J$, where signals between near neighbors can be exchanged in one time step. Request packets (RPs) are submitted to C^1. Each C^j, with $j < J$, is connected to C^{j+1} by an *upward* link. Upward links establish a path, referred to as the *U-path*, along which an RP directed at location x travels up to controller $C^{\ell(x)}$ and is submitted to level $\mathcal{M}^{\ell(x)}$. Since RPs enter only at C^1 and require only constant amount of processing at each level, the U-path can be trivially pipelined to accept one RP per time step ($p(M) = 1$, independent of M). The logic required at each controller to implement the U-path is indeed straightforward.

Each C^j, with $j > 1$, is also connected to C^{j-1} by a *downward* link and C^1 is connected to a global memory queue, MQ, in order to establish a *D-path* along which an RP serviced at $\mathcal{M}^j$ travels from the output of Q^j to MQ. Designing an efficient pipelined D-path is a more challenging task, since RPs can enter at any level. Our objective is to bound both the waiting time in Q^j and the time spent to travel from level j to MQ once in the D-path. Indeed, we will bound the D-path time by $O(j)$ and the waiting time in Q^j by a function of the level latencies $\mu^1, \ldots, \mu^j$, a function which becomes $O(j)$ when the latencies are suitably chosen.

Our design of the D-path carefully controls the available bandwidth to ensure finite progress, both for the "new" RPs that are injected into the D-path at various levels and for the "old" RPs that are already traveling down the D-path. For injection of new RPs into the D-path, lower levels are designed to have precedence over higher levels, to balance the response times. This is accomplished by adjusting two integer parameters, w and b, with $w > b > 0$, as depicted in Figure 3. Each downward link has bandwidth w, in RPs per time step. Every C^j is equipped with two sets of buffers B^{j0} and B^{j1}, each capable of holding b RPs. Set B^{j0} will be updated only in time steps of

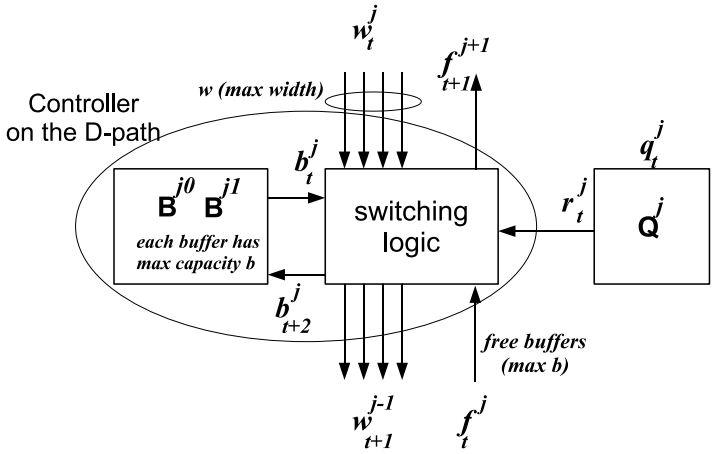


Figure 3: Schematic view of the D-Path within controller C^j, with its input, state, and output variables. The switching logic implements Equations (1)-(5).

even parity, while set B^{j1} will be updated only in time steps of odd parity. Each buffer is organized as a FIFO queue; if, during the same time step, RPs are received both from Q^j and from C^{j+1}, the RPs from C^{j+1} are enqueued first. Informally, the performance role of the two parameters is as follows: at least b units of bandwidth are used to guarantee access of "new" RPs into the D-path and at least $w-b$ units are used to guarantee progress to the "old" RPs within the D-path. When the old RPs do not use up all of their bandwidth allowance, the excess can be used by the new ones and vice versa. This behavior is achieved with the help of a control signal, f_t^j, sent to C^j by C^{j-1}, representing the bandwidth guaranteed for new RPs.

At any time step t, the following quantities determine the behavior of C^j: the number b_t^j of occupied buffers in set $B^{j,t \bmod 2}$, the number w_t^j of RPs received from C^{j+1}, the control variable f_t^j, set by C^{j-1} at time $t-1$ with $f_t^1 = b$ for all $t's$, and q_t^j defined as the number of RPs ready to enter the D-path, which include those left in Q^j at the end of time step $t-1$ as well as the RP exiting $\mathcal{M}^j$ at $t-1$, if any. As a function of these variables, the following quantities are set by C^j at time t: the number r_t^j of RPs to be accepted from Q^j; the number w_{t+1}^{j-1} of RPs to be sent down to C^{j-1}; the control signal f_{t+1}^{j+1} to be sent up to C^{j+1}, and the number b_{t+2}^j of RPs to be occupied in set $B^{j,t \bmod 2}$. Note that, since each $B^{j,t \bmod 2}$ is accessed in alternate steps, the updated number is b_{t+2}^j, rather than b_{t+1}^j. We assume that at $t=0$ the D-path is empty, that is, $b_0^j = b_1^j = 0$ and $w_0^j = 0$ for all j's. In addition, we let $f_0^j = b$, for all j's. The rules governing the behavior of C^j, for $t \geq 0$, are stated next and discussed below.

$$r_{max,t}^j = f_t^j + [w - (w_t^j + b_t^j)], \quad (1)$$

$$r_t^j = \min(q_t^j, r_{max,t}^j), \quad (2)$$

$$w_{t+1}^{j-1} = \min(r_t^j + w_t^j + b_t^j, w - b + f_t^j), \quad (3)$$

$$b_{t+2}^j = r_t^j + w_t^j + b_t^j - w_{t+1}^{j-1}, \quad (4)$$

$$f_{t+1}^{j+1} = b - b_{t+2}^j. \quad (5)$$

To illustrate the role played by the above rules, it is helpful to realize that they preserve the

following two invariants, which trivially hold at times $t = 0, 1$:

$$\text{Invariant 1 :} \qquad w - (w_t^j + b_t^j) \geq 0, \tag{6}$$

$$\text{Invariant 2 :} \qquad 0 \leq f_t^j \leq b. \tag{7}$$

Rule (1) sets a threshold $r_{max,t}^j$ to the number of RPs that can transfer from Q^j into the D-path. The term f_t^j represents the bandwidth reserved to new RPs; the term in square brackets, which due to Invariant 1 is never negative, represents the bandwidth not used by old RPs that is made available to the new ones. Rule (2) states that number of RPs that actually transfer from Q^j into the D-path is determined by the threshold or by the size of Q^j, whichever is smaller.

Rule (3) states that up to $w - b + f_t^j$ of the RPs that C^j respectively holds in the buffers (b_t^j), receives from above (w_t^j), and accepts from the queue (r_t^j) are sent below to C^{j-1}. The remainder, if any, are stored in the buffers, as reflected by Rule (4).

It is easy to see (from Rules (3) and (4)) that when $r_t^j + w_t^j + b_t^j \leq w - b + f_t^j$, one has $w_{t+1}^{j-1} = r_t^j + w_t^j + b_t^j$ and $b_{t+2}^j = 0$. Otherwise, $w_{t+1}^{j-1} = w - b + f_t^j$ and, by Rules (4) and (1),

$$b_{t+2}^j = r_t^j + w_t^j + b_t^j - w + b - f_t^j \leq r_{max,t}^j + w_t^j + b_t^j - w + b - f_t^j = b.$$

Thus, $b_{t+2}^j \leq b$, which both ensures that the buffers can hold the required number of RPs and, in view of Rule (5), preserves Invariant 2.

Finally, Rule (3) for C^{j+1} at time $t + 1$ and Rule (5) for C^j at time t yield

$$w_{t+2}^j \leq w - b + f_{t+1}^{j+1} = w - b_{t+2}^j,$$

thus establishing Invariant 1 at time $t + 2$ for C^j.

The property established next sheds some light on the behavior of the D-path by showing that a decrement $f_t^j - f_{t+1}^{j+1}$ of what we have intuitively indicated as the bandwidth guarantee for new RPs does indeed correspond to at least that many RPs entering the D-path from Q^j at time t.

Proposition 2 *For any t and j, we have*

$$r_t^j \geq f_t^j - f_{t+1}^{j+1}. \tag{8}$$

Proof: Bound (8) is obtained by the following chain of relations, starting from (5) and making subsequent use of (4), (3), some algebra, Invariant 1, and Invariant 2.

$$\begin{aligned}
f_{t+1}^{j+1} &= b - b_{t+2}^j \\
&= b - (r_t^j + w_t^j + b_t^j) + w_{t+1}^{j-1} \\
&= b - (r_t^j + w_t^j + b_t^j) + \min(r_t^j + w_t^j + b_t^j, w - b + f_t^j) \\
&= \min(b, w + f_t^j - r_t^j - w_t^j - b_t^j) \\
&\geq \min(b, f_t^j - r_t^j) \\
&= f_t^j - r_t^j.
\end{aligned}$$

□

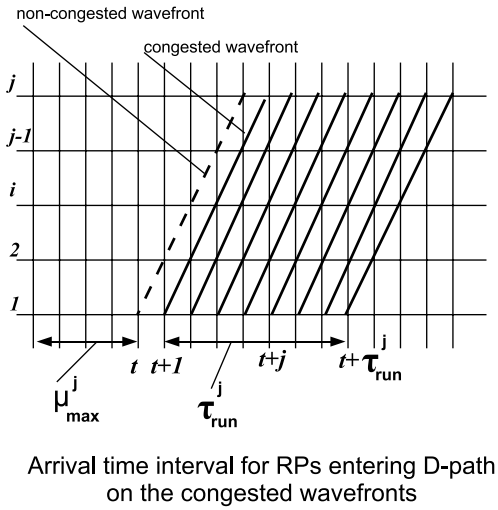


Figure 4: Space-time diagram depicting waves of RPs entering the D-path.

3.4 Analysis of Access Time

Next, we analyze the access time, λ^j, for a read RP whose target location is at level $\mathcal{M}^j$. This time can be divided into four components: (i) the time, $up^j = j$, spent to go up the U-path to reach $\mathcal{M}^j$; (ii) the service time, μ^j, spent in $\mathcal{M}^j$; (iii) the queueing time, τ^j, spent in Q^j before entering the D-path, which we will bound below; and (iv) the time, $down^j$, spent in the D-path. Component (iv) is bounded by $2\lceil\frac{b}{w-b}\rceil j$. To see this, consider that the RP must go through j controllers in the D-path, each controller has a pair of buffers operated in alternate time steps, and each buffer contains at most b RPs, at least $w-b$ of which move to the next controller every second step. Thus, we are lead to the bound

$$\begin{aligned} \lambda^j &= up^j + \mu^j + \tau^j + down^j & (9)\\ &\leq j + \mu^j + \tau^j + 2\lceil\frac{b}{w-b}\rceil j. & (10) \end{aligned}$$

Our next goal is to bound τ^j. Let us consider a time-space diagram of Figure 4, where a point (t,i) is associated with time step t and level i. With reference to a fixed level j, the diagonal line in Figure 4 connecting the points $(t+i,i), i=1,\ldots,j$, is called the *wavefront* based at time step t. We let $R_t^j = \sum_{i=1}^{j} r_{t+i}^i$ denote the total number of RPs that enter the D-path along the points of the wavefront. We say that a wavefront is *congested* if for some level h, with $1 \leq h \leq j$, the queue Q^h does not get emptied at time $t+h$, *i.e.*, not all the q_{t+h}^h RPs left over in Q^h from time $t+h-1$ or just arrived at time $t+h$ manage to enter the D-path at time $t+h$. We now show that, on a congested wavefront, at least b RP's enter the D-path. Formally, we have:

Proposition 3 *For any t, if the wavefront based at time t is congested, then $R_t^j \geq b$.*

Proof: Let $h \leq j$ be a level where the queue does not get emptied at time $t+h$. By summing Equations (8) for $i=1,\ldots,h$, and recalling that $f_t^1 = b$, for all t, we get

$$R_t^j \geq R_t^h \geq f_{t+1}^1 - f_{t+h+1}^{h+1} = b - f_{t+h+1}^{h+1}. \quad (11)$$

The fact that Q^h does not get emptied implies that, in Equation (2), we have $r^h_{t+h} = r^h_{max,t+h}$; substituting this in Equations (3), (4), and (5), we get $f^{h+1}_{t+h+1} = 0$. Combining this with Equation (11) we obtain the stated result. □

For a given level j, a *run* of length τ^j_{run} is a sequence of congested wavefronts based at times $(t+1, \ldots, t+\tau^j_{run})$, preceded by a *non*-congested wavefront based at time t. It is clear that the τ^j time steps during which an RP is queued in Q^j must be part of a run, whence an upper bound on the length of any run is also an upper bound on the waiting time τ^j. Therefore, we turn our attention to the length of a run.

Let RP_{run} be the set of RPs entering the D-path on any of the τ^j_{run} congested wavefronts of a run based at times in the interval $(t+1, t+\tau^j_{run})$. By the definition of run and by Proposition 3, we have $|RP_{run}| \geq b\tau^j_{run}$. Denoting $\mu^j_{max} = \max_{i=1}^{j} \mu^i$, any RP entering the D-path on any of the above congested wavefronts must have been submitted in the time interval, $(t+1-\mu^j_{max}, t+\tau^j_{run})$, which is of length $\mu^j_{max}+\tau^j_{run}$ as depicted in Figure 4. This is because if an RP is submitted earlier than this interval, then it must have entered the D-path on or before the non-congested wavefront based at t. Similarly if it is submitted later than the interval, it cannot enter the D-path on any of the wavefronts of the run. Since only one RP can be submitted in any time step, we conclude $\mu^j_{max} + \tau^j_{run} \geq |RP_{run}| \geq b\tau^j_{run}$, which gives us $\tau^j_{run} \leq \mu^j_{max}/(b-1)$. Thus, we get the following bound for queueing delay:

Proposition 4 *The time spent by any RP in queue Q^j satisfies*

$$\tau^j \leq \mu^j_{max}/(b-1). \tag{12}$$

Combining this proposition with relation (9), we arrive at the key result of this section.

Theorem 1 *In the proposed* ***pipelined hierarchical memory*** *organization read and write requests can be submitted one per step, that is, $p(M) = 1$. A read request directed to level j is satisfied in time*

$$\lambda^j \leq j + \mu^j + \mu^j_{max}/(b-1) + 2\lceil \frac{b}{w-b} \rceil j. \tag{13}$$

Of particular interest to us is the case of d-dimensional geometries, considered in the next corollary.

Corollary 1 *For a d-dimensional memory where the level $\mathcal{M}^j$ is realized as in Section 3.1, with size $M^j = (j)^{d-1}$ and latency $\mu^j = (d-1)j$, and choosing for the backbone $w = 4$ and $b = 2$, a read RP directed to location x is satisfied at most in time*

$$a(x) = O(dx^{1/d}). \tag{14}$$

Proof: First, notice that with the choice of parameters under consideration, (13) can be rewritten as

$$\lambda^j \leq j + 2(d-1)j + 2(d-1)j + 2j = (4d-1)j. \tag{15}$$

Since the collective size of the first j memory levels is $\sum_{i=1}^{j} j^{d-1} = \theta(j^d)$, clearly location x is at a level $\ell(x) = O(x^{1/d})$. Using the latter relation in (15) yields (14). □

While the choice $w = 4$ and $b = 2$ for the D-path parameters yields the desired proportionality of $a(x)$ to $x^{1/d}$, other choices are possible, yielding different bandwidth/access-time tradeoffs. Similarly, there are different choices for the memory levels $\mathcal{M}^1, \mathcal{M}^2, \ldots$ To mention just one example, one could choose to let $\mathcal{M}^j$ be empty except when j is a power of two and, in such case, let $\mathcal{M}^j$ be a d-dimensional memory of size $M^j = j^d$. The exploration of the corresponding tradeoffs would definitely be of interest in a phase of engineering optimization of the system.

3.5 Memory Queue

Referring to Figure 2, the incoming packets on the U-path can be either read or write requests generated by an instruction, whereas the outgoing packets on the D-path can only be read packets. The latter packets are shown to be entered into a queue called MQ. In a complete machine, they will be subsequently consumed by the processor. We assume that MQ has the following properties: (i) it implements a FIFO policy; (ii) it can accept up to w packets per step (as many as can be output by the D-path); (iii) it can output one packet per step, as long as it is not empty. Implementations of MQ that are extensible under physical constraints, as specified in Section 2.3, can be developed, for example with a systolic-like approach [27]. The detailed design is an exercise, although not completely trivial.

One issue that arises is the maximum size of the queue for which the physical implementation must be tailored. In general, this size depends upon how the processor extracts entries from the queue. As an illustration, let us consider the case of a processor that always extracts the head of the queue if the latter is not empty. Further assume that the pipelined hierarchical memory (PHM) has size M, period $p(M) = 1$, and access function $a(x)$. Then, we claim that (1) the time spent by an RP waiting in MQ equals the size of the queue upon the arrival of the RP, and (2) the size of MQ is bounded above by $a(M)$, at any time. Property (1) is an obvious consequence of the queue being depleted at the rate of one entry per step. To establish Property (2), consider the combined packet population in both the PHM and the MQ. At time steps when MQ is non-empty, the population does not increase, since one packet is definitely dequeued from MQ and at most one packet can be submitted to the PHM. On the other hand, at times when MQ is empty, the population is completely contributed by the PHM and cannot be larger than $a(M)$, since at most one packet per step can enter the PHM and no packet stays in the PHM longer than $a(M)$ steps.

4 Time Lower Bounds for a Class of Machines

It is well known that, for machines with a hierarchical memory that does not support any access concurrency, some computations of N instructions do require time superlinear in N, for many reasonable access functions. Results of this type, for computations such as semi-ring matrix multiplication, Fast Fourier Transform, and sorting, can be found in [2], where the Hierarchical Memory Model (HMM) of computation is introduced. The arguments are based on a lower bound technique pioneered by [24] and further developed in [35, 15].

The ability to support concurrent accesses is clearly beneficial to performance. The Block Transfer (BT) model of [3] goes in this direction, assuming the availability of a block-move instruction that can copy the content of a set of k consecutive memory locations into another such set, in time $a(x_{max}) + k$, where x_{max} is the largest address of any source or target location. A number of algorithms can be implemented considerably faster in BT than in HMM. Yet, some very basic computations, *e.g.,* incrementing all the components of a given array, do take superlinear time in the number of instructions in BT, as established in Theorems 2.1 and 5.1 of [3]. It is interesting to observe that the cited lower bounds apply to computations with no data (functional or address) dependences, thus essentially exposing a limitation of the ability to pipeline accesses only from and to consecutive addresses.

The Pipelined Hierarchical RAM (PH-RAM) was proposed in [9] to circumvent the limitations of block transfer. The PH-RAM supports pipelined moves even when the source (or destination) addresses are not consecutive. It also allows concurrency among memory accesses performed by

different data-processing instructions. The PH-RAM can simulate the BT model with constant slowdown and is also faster than BT on some computations. As an example, it can increment all the components of a given array in linear time.

However, even the availability of a memory with constant pipeline period, postulated by the PH-RAM and shown to be feasible in Section 3, is not sufficient to guarantee full access overlap, even for the restricted case of direct-flow programs, whose accesses are all statically determined. While high memory concurrency is a crucial factor, overall machine performance also requires suitable processor organizations capable to take advantage of the memory pipeline. We will propose two such processor organizations in Section 5.

To develop insights into the issues involved, in this section, we explore a class of system organizations satisfying a few assumptions which, while quite natural, ultimately limit the achievable performance. We establish that certain computations require time superlinear in the number of executed instructions, on any system of the defined class.

The definition of our class of machines involves a number of conditions, and their relevance may not be immediately clear. However, the class does include several computational models previously considered in the literature, as well as some models for commercial machines, as discussed below.

4.1 Machines with Explicit Data Movement

We now introduce and discuss the assumptions that define the class of machine organizations for which lower bound arguments are developed in the next subsection. We refer to this class as explicit data movement or EDM machines.

A1. *Physical Addressing.* At any time, the data-state of the program is stored in a set of physical locations addressable by the non-negative integers ($x = 0, 1, \ldots$). The instruction set is of the type specified in Section 2.1. Instructions refer to data using physical addresses.

A2. *Memory Management by Program Only.* The state of the physical locations is modified exclusively by the execution of data-processing instructions of the form $< opcode, x, y, z >$, whereby the content of location z is updated with the value of a suitable function (specified by *opcode*) of the contents of locations x and y. Data-processing instructions can be either *ALU-instructions* or *copy-instructions*. The former require an arithmetic/logical unit (ALU) to transform the source operands into a result operand, whereas a copy-instruction simply copies a single source operand to the destination. Thus, a copy instruction is of the form $< copy, x, z >$, with the obvious interpretation.

A3. *Literal Execution.* The execution of an instruction with an operand (or condition, or target) specified by a direct or an indirect addressing mode does generate the submission of the corresponding read request(s) to the memory. Similarly, the execution of an instruction with a result does generate the submission of the corresponding write request to the memory. We say that an instruction is *issued* when it submits the first request to fetch its input operands and *committed* when it submits the request to store its result into memory. We say that an instruction is *in flight* from the time it is issued to the time it commits. There is a non-decreasing, positive integer function $a(x)$, called the *access function* of the system, such that an instruction $< opcode, x, y, z >$ is in flight for at least $\max(a(x), a(y))$ time.

A4. *Memory Sequential Consistency.* A read request submitted to memory for a given location always returns the value written by the most recent write request submitted to that location prior to the read in question.

A5. *Serial, Inorder Issue.* Instructions submit the read requests corresponding to their operands one at the time, to memory, in program order.
A6. *Read-after-Write.* An instruction with an operand at location x can be issued only after all the instructions preceding it in program order that write into location x have committed their result to their target location.
A7. *Memory Concurrency.* At any given time, at most s copy-instructions are in flight.
A8. *ALU Concurrency.* At any given time, at most f ALU-instructions are in flight.

Assumptions A1-A3 essentially constrain the components of the physical state of the machine reflecting the architectural state of storage to be tightly controlled by the program. In particular, A3 models the fact that, when executing an instruction, the operand information is actually retrieved from the physical locations *literally* specified by that instruction. In contrast, one may observe that Assumptions A1-A3 are not typically satisfied by machines where addresses are regarded as names of virtual locations and get dynamically mapped to physical locations. Examples of such machines are those with cache management and register renaming mechanisms. While, for simplicity, the lower bounds below will be derived under assumptions A1-A3, we will also briefly indicate how the argument could be adapted to cover forms of virtual memory management not too different from those encountered in most current commercial systems.

Assumptions A4-A6 are sufficient, although not necessary, to correctly implement the program semantics. The following simple corollary of A5 and A6 has far reaching consequences for performance:

Principle of Non-Overlapping Flight. Let us say that a write operation performed by an instruction is *useful* if the written value is read by at least one subsequent instruction. Then, two instructions that perform useful writes to the same location cannot be in flight at the same time.

Assumptions A7 and A8 are simply introducing parameters that allow us to investigate the impact of limiting the concurrency of data movement between physical locations (s) and toward the ALU (f). For example, consider a RISC architecture, where all data copies must go through registers (which, in terms of Assumption A1, are part of the set of physical locations). Then, owing to the principle of non-overlapping flight, s is upper bounded by the size of the register file.

The Assumptions A1-A8 are satisfied by a number of computational models, for suitable choices of $a(x)$, s and f. These include the HMM [2] and the PH-RAM [9]. The BT model [3] does not strictly fall in our machine class, as its block-transfer instruction is outside the instruction set introduced in Section 2, since it can affect a non constant number of memory locations. Nevertheless, it is straightforward to simulate BT by the PH-RAM with constant slowdown. Hence, the lower bounds to be derived next apply to all of these models.

4.2 A Lower Bound Technique

We will formulate our lower bounds for computations specified as computation directed acyclic graphs (CDAGs), where nodes model operations as well as input data (nodes with no incoming arc) and arcs model functional dependence. For a more formal definition, the reader can refer, *e.g.,* to [14]; however, the informal notion ought to be sufficient here. CDAGs can be implemented as direct-flow programs. One degree of freedom of the implementation is the schedule of the operations, which can be chosen according to any topological sorting of the DAG. Another degree is the memory map, that is, the assignment of a memory location to inputs and to results of instructions. This

assignment can vary dynamically, by means of copy operations, to take advantage of the hierarchical nature of memory.

The time lower bound we will derive for a CDAG G applies to any program implementing G. The lower bound depends upon a CDAG metric c, introduced in the next proposition, which limits the amount of parallelism available at any stage of the execution of G.

Proposition 5 *Let $G = (V, E)$ be a CDAG. We say that G is c- **coverable** if its operation nodes can be covered (possibly, with overlap) by c directed linear chains. Then, throughout the execution of any program that computes a c-coverable G on any EDM machine, at most c ALU instructions can be concurrently in flight.*

Proof: Given a set of more than c ALU instructions, by the pigeon-hole principle, at least two of them, say I_h and I_k (with $h < k$), must correspond to CDAG nodes that lie in the same chain, γ.

Suppose first that the nodes corresponding to I_h and I_k are consecutive on γ. Then, there is a value v written by I_h in some location x_0, possibly copied by intermediate instructions $I_{h_1}, \ldots, I_{h_j}$ into locations $x_1, \ldots, x_j$, and finally read by I_k from location x_j. Except for I_h, each instruction in the subsequence $I_h, I_{h_1}, \ldots, I_{h_j}, I_k$ reads a location written by the previous one. Then, by Assumption A6, each instruction can be issued only after the previous one has committed. We conclude that no two instructions in the subsequence, in particular I_h and I_k, can be simultaneously in flight.

A simple inductive argument on d extends the proof to the general case when I_h and I_k are at distance d on chain γ. □

The exact technical formulation of the lower bound derived below is rather involved, due to (a) the number of parameters that play a role, (b) the generality of the access function, and (c) the nature of the underlying counting argument. However, the main thrust is that EDM machines cannot always execute a computation in time proportional to the number of required operations, even when the memory is fully pipelinable. Limitations to the amount of pipelinability can further reduce performance. The fact that some direct-flow programs satisfy superlinear lower bounds indicates how the performance bottleneck is due to inherent limitations in orchestrating the data movement under Assumptions A1-A8.

To help the reader follow the technical developments, we first provide some intuition behind the lower bound proof. If the computation has N input values, at least $N/2$ of them will initially be placed at addresses not smaller than $N/2$. These "distant" inputs must eventually arrive at the processor, possibly after being copied into temporary intermediate memory locations. In view of Proposition 5, at most c operation instructions can be concurrently in flight; since c could be as small as 1 for suitable computation graphs, a program based only on ALU instructions can result in $N/2$ non-overlapping accesses each taking time at least $a(N/2)$. The advantage of intermediate copying stems from the capability of pipelining many copy instructions, with a substantial overlap of their access latencies; thus, many, say k, values can be brought in a faster memory region, say within the g smallest locations, in a relatively short time, and accessed later at a smaller cost. Since the flight time of two copy instructions with the same target location cannot overlap, one must have $k \leq g$. Then, the larger is the number k of values concurrently copied, the larger is the latency of subsequent accesses to (a substantial fraction of) these values at the new locations. On the other hand, intuitively, k ought to be chosen to be of the same order as the maximum latency of the source addresses involved in the bulk copy so that k values can be moved in $O(k)$ time. The bulk copy approach can be deployed recursively, until the input values are brought at a suitably small distance from the processor, which can then access them directly. As the number of levels in this recursive process increases, the total amount of non-overlapped latency tends to decrease at the

expense of a larger number of copy instructions which, owing to rule A5 on serial issue, sets a lower bound to the computation time.[‖] To capture this tradeoff for a program implementing a given computation graph, we consider an arbitrary partition of the memory into regions of consecutive addresses. We then develop lower bounds to the execution time both for the case when each region is the target of copy operations for many input values (here, the bound stems from the total number of copy instructions) and for the complementary case when there is at least one region not targeted by many copy operations (here the bound stems from the total memory latency that cannot be overlapped). The resulting lower bound depends upon the number of regions in the partition as well as upon their boundaries. The partition that yields the best bound varies with the $a(x)$ function, as illustrated in a few corollaries of the main theorem. Next, we introduce some notation instrumental to a concise statement of the lower bound.

Definition 6 *For any positive integer k, let $Y(k)$ denote the set of the increasing sequences of positive integers not greater than k,* i.e.*:*

$$Y(k) = \cup_{L=1}^{k} \{(y_1, \ldots, y_L) : \ 0 < y_1 < \ldots < y_L \leq k\}.$$

Given access function $a(\cdot)$, $\mathbf{y} = (y_1, \ldots, y_L) \in Y(k)$, and positive integers s and q, we let

$$\lambda_{a(\cdot)}(\mathbf{y}; s, q) = \min\{2L, 2a(y_1), a(y_2)/\max(q, \min(s, y_1)), \ldots, a(y_L)/\max(q, \min(s, y_{L-1}))\},$$

$$\Lambda_{a(\cdot)}(k; s, q) = \max_{\mathbf{y} \in Y(k)} \lambda_{a(\cdot)}(\mathbf{y}; s, q).$$

Informally, the notation will be used as follows. The integer vector $\mathbf{y} = (y_1, \ldots, y_L)$ defines the boundaries of the regions in a memory partition. The set $Y(k)$ encodes a set of partitions; the proof will focus on $Y(N/2)$. For $\mathbf{y} \in Y(N/2)$, the function $\lambda_{a(\cdot)}(\mathbf{y}; s, q)$, the shape of which emerges from the structure of the counting argument, denotes a lower bound to T/N, based on the memory partition defined by $\mathbf{y}$. The parameters s and q respectively denote the memory and the alu concurrency afforded by the machine as well as by the computation. Since the lower bound holds for any $\mathbf{y} \in Y(N/2)$, one is interested in maximizing with respect to $\mathbf{y}$, whence the role of $\Lambda_{a(\cdot)}(k; s, q)$. In spite of its apparent intricacy, the function $\Lambda_{a(\cdot)}(N/2; s, q)$ can be actually evaluated for a number of interesting cases, as illustrated by a few corollaries, following the main theorem.

Theorem 2 *Let M be a machine satisfying Assumptions A1-A8, with parameters $a(\cdot)$, s, and f. Let G be a c-coverable CDAG with N inputs, $u_1, u_2, \ldots, u_N$, available in memory at the beginning of execution. Let $q = \min(c, f)$. The time to execute G on M satisfies the bound*

$$T_{a(\cdot)}^{s,q}(G) \geq (N/8) \max(4a(N/2)/(s + 2q), \Lambda_{a(\cdot)}(N/2; s, q)). \tag{16}$$

Proof: As a preliminary observation, we consider that $q = \min(c, f)$ is an upper bound on the number of ALU-instructions that can be simultaneously in flight, where the limit c arises from the nature of the computation (Proposition 5) and the limit f arises from a machine constraint (A8).

[‖] The interested reader can find these ideas applied to the design of efficient algorithms for the BT model [3] and for the PH-RAM model [9]. A systematic technique to optimize the number and the size of bulk copying at various levels of recursion is also presented in [9].

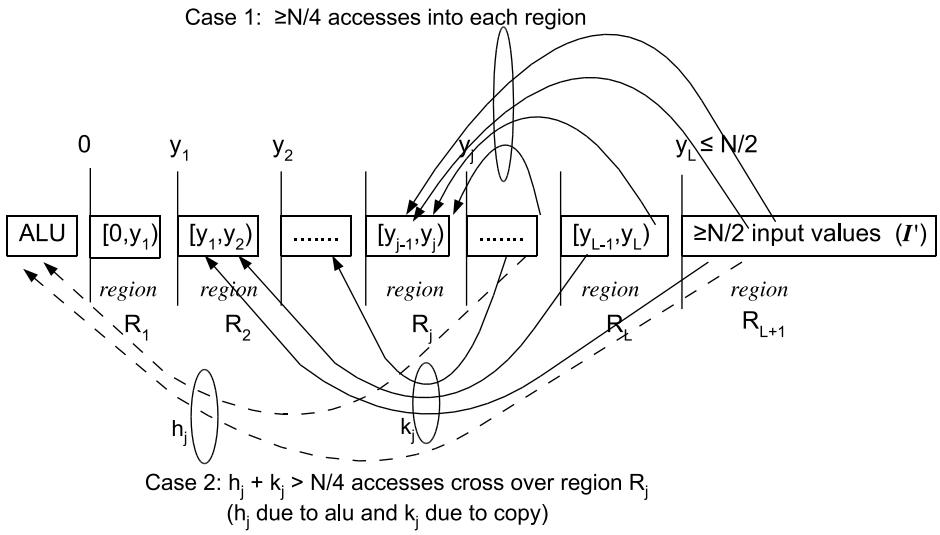


Figure 5: Illustration of the counting argument for lower-bound analysis.

We first argue that $T^{s,q}_{a(\cdot)}(G) \geq (N/8)4a(N/2)/(s+2q) = (N/2)a(N/2)/(s+2q)$. Simply consider the $(N/2)$ inputs initially located at the highest addresses. All of them must be accessed at least once, for a total latency non smaller than $(N/2)a(N/2)$. It remains to observe that at most $(s+2q)$ accesses can overlap: at most s due to copy-instructions (which have only one operand) and at most q due to ALU-instructions (which may have up to two operands).

We next argue that $T^{s,q}_{a(\cdot)}(G) \geq (N/8)\Lambda_{a(\cdot)}(N/2; s, q)$. Consider any $\mathbf{y} = (y_1, \ldots, y_L) \in Y(N/2)$ and ideally partition the address space into the $L+1$ regions $R_j = [y_{j-1}, y_j)$, for $j = 1, \ldots, L+1$, with the convention that $y_0 = 0$ and $y_{L+1} = \infty$. At the beginning of the execution, a set I' of at least $N/2$ input values must reside in region R_{L+1}, since at most $N/2$ of the N input values can be stored in regions $R_1, \ldots, R_L$, (by definition, $y_L \leq N/2$). During execution, each $u_i \in I'$ is accessed at least once during the computation, since it is the operand of at least one operation. Other accesses involving value u_i can also occur, for instance because of copy instructions. For $j = 1, \ldots, L$, let c_j be the number of elements in I' such that (a) a copy of the element is written into a location within R_j at some stage of the execution and (b) that copy is subsequently read from that location. Consider the following two complementary cases.

Case 1: Many copies in each region. If for each $j = 1, \ldots, L$ we have $c_j \geq N/4$, then there must have been at least $N/4$ copy instructions with target in each region R_j, taking a total of at least $(N/4)L$ time steps, in view of Assumption A5. Thus, the time taken for case 1 must be at least

$$T_1 \geq (N/8)(2L). \tag{17}$$

Case 2: Few copies in at least one region. Otherwise, let j be such that $c_j < N/4$. Then, more than $N/4$ of the elements of I' are read at least once from a location $x \geq y_j$. These readings are said to be of category (i) when the element is an operand of an ALU-instruction and of category (ii) when the element is the operand of a copy-instruction with target in $R_1 \cup R_2 \ldots \cup R_{j-1}$. Let us say that h_j elements fall in category (i) and k_j fall in category (ii), where obviously $h_j + k_j > N/4$. (Observe that $k_1 = 0$.) Next, we consider the time needed to process each category.

Claim 1. The time spent to process category (i) satisfies

$$t'_j \geq h_j a(y_j)/q.$$

Indeed, each of the h_j operations reading an input from a location $x \geq y_j$ will take at least $a(x) \geq a(y_j)$ units of time, by Assumption A3. Then, the bound stated on t'_j follows considering that at most q such operations can be simultaneously in flight.

Claim 2. The time spent to process category (ii), which only arises for $j \geq 2$, since $k_1 = 0$, satisfies

$$t''_j \geq k_j a(y_j)/\min(s, y_{j-1}).$$

In any execution of CDAG G, at any given time, of the k_j copy-instructions with target in $R_1 \cup R_2 \ldots \cup R_{j-1}$, there can be at most one in flight having as target a given location x. This is simply an application of the non-overlapping flight principle, derived earlier in the section from Assumptions A5 and A6. Since there are y_{j-1} distinct locations in $R_1 \cup R_2 \ldots \cup R_{j-1}$, the number of category (ii) instructions simultaneously in flight is at most y_{j-1}. But, by A7, the same number is also at most s, hence it is at most $\min(s, y_{j-1})$.

Since each of the k_j copy instructions under consideration reads the input from a location $w \geq y_j$, it will be in flight for at least $a(w) \geq a(y_j)$ units of time (by Assumption A3) which, together with the upper bound on concurrency just derived, yields the bound stated on t''_j.

Let T_{2j} denote the time for case 2 when R_j is the region having $c_j < N/4$. Combining the bounds of the two claims, for $j = 1$, considering that $h_1 > N/4$, we have

$$T_{21} > (N/4)a(y_1)/q = (N/8)2a(y_1)/q,$$

while for $j = 2, 3, \ldots, L$ we have

$$\begin{aligned} T_{2j} &\geq \max(t'_j, t''_j) \geq \max(h_j a(y_j)/q, k_j a(y_j)/\min(s, y_{j-1})) \\ &\geq [(h_j + k_j)/2] \;\; [\min(a(y_j)/q, a(y_j)/\min(s, y_{j-1}))] \\ &> [N/8] \quad\; [a(y_j)/\max(q, \min(s, y_{j-1}))]. \end{aligned}$$

Since either Case 1 occurs or else Case 2 must occur for some $j \in \{1, 2, \cdots, L\}$, the execution time of G satisfies at least one of the corresponding bounds, that is:

$$T^{s,q}_{a(\cdot)}(G) \geq \min(T_1, T_{21}, \ldots, T_{2L}) \geq (N/8)\lambda_{a(\cdot)}(\mathbf{y}; s, q),$$

where we have used the preceding bounds on the various $T's$, as well as Definition 6 for $\lambda_{a(\cdot)}(\mathbf{y}; s, q)$. Finally, since the derived bound holds for every $\mathbf{y} \in Y(N/2)$, it holds in particular for the choice of $\mathbf{y}$ that maximizes the right hand side which, recalling Definition 6 for $\Lambda_{a(\cdot)}(N/2; s, q)$, yields the desired bound. □

The meaning of bound (16) is not immediately intuitive. In general, the max term can give rise to superlinear time. If the CDAG G has $\theta(N)$ nodes, then, to within constant factors, the max term in (16) is a lower bound to the CPI. An appreciation for the lower bound can be developed by considering a number of interesting special cases, where a simple closed form expression can be derived. A few such examples are developed in the following corollaries, with the notation and the assumptions of Theorem 2.

Corollary 2 *If $q = \theta(a(N))$ and $s = O(1)$, then $T = \Omega(N)$.*

Informally, this straightforward corollary states that, if the operation concurrency is of the same order as the maximum memory latency, no superlinear effect arises, even if copy concurrency is limited to a constant level.

Corollary 3 *If $q = O(1)$ and $s = \theta(a(N))$, we have:*

1. *If* $a(x) = \theta(x)$, *then* $T = \Omega(N \log N / \log\log N)$.

2. *If* $a(x) = \theta(x^\alpha)$, *with* $0 < \alpha < 1$, *then* $T = \Omega(N \log\log N)$.

3. *If* $a(x) = \theta(\log x)$, *then* $T = \Omega(N \log^* N)$.

Proof: First, we derive from Bound 16 the simpler bound

$$T \geq \Omega(N \lambda_{a(\cdot)}(\mathbf{y}; s, q)), \tag{18}$$

holding for any $\mathbf{y} \in Y(N/2)$. Second, we observe that $q = O(1)$ yields $\max(q, \min(s, y_j)) = O(\min(s, y_j))$. Below, we will choose a $\mathbf{y}$ such that $y_{L-1} = O(a(N))$. With such a choice, in view of the assumption $s = \theta(a(N))$, for $j \leq L-1$, we have that $\min(s, y_j) = O(y_j)$. Using the foregoing relations in Definition 6, we obtain

$$\lambda_{a(\cdot)}(\mathbf{y}; \theta(a(N)), O(1)) = \Omega(\min\{2L, 2a(y_1), a(y_2)/y_1, \ldots, a(y_L)/y_{L-1}\}). \tag{19}$$

Below, for each access function $a(x)$ considered in the statement, we define the y_j's as functions of N and j and, correspondingly, let $L = \max\{j : y_j \leq N/2\}$. Plugging the consequent bounds for $L, a(y_1), a(y_j)/y_{j-1}$ in Relations 19 and 18, yields the stated time lower bounds.

1. $y_j = (\log N / \log\log N)^j$, which implies

$$L, a(y_1), a(y_j)/y_{j-1} = \Omega(\log N / \log\log N).$$

2. $y_j = (\log\log N)^{e_j}$ where $e_j = ((1/\alpha)^j - 1)/(1-\alpha)$, which implies

$$L, a(y_1), a(y_j)/y_{j-1} = \Omega(\log\log N).$$

3. $y_j = 2^{y_{i-1} \log^* N}$, with $y_1 = 2^{\log^* N}$, which implies

$$L, a(y_1), a(y_j)/y_{j-1} = \Omega(\log^* N).$$

The tedious, but straightforward calculations needed to verify the stated bounds on $L, a(y_1), a(y_j)/y_{j-1}$ and the above claimed relation $y_{L-1} = O(a(N))$ are omitted. □

Informally, the preceding corollary tells us that, if the operation concurrency is limited to a constant level, then, even if the maximum copy concurrency is allowed, there is a superlinear effect for all non-constant access functions, the effect being larger for faster growing functions.

The corollary applies, for example, to a CDAG consisting of just one chain, where the first operation takes as operands input values u_1 and u_2 and, for $i = 2, \ldots, N-1$, the i-th operation takes as operands the result of the $(i-1)$-st and input value u_{i+1}. This CDAG is 1-coverable ($c = 1$). Matching upper bounds for computing a chain graph in the PH-RAM model (a machine satisfying assumptions A1-A8) are given in [9].

Corollary 4 *If* $q = O(1)$ *and* $s = O(1)$, *then* $T = \Omega(Na(N))$.

Informally, this straightforward corollary provides the bound for non-pipelinable memory hierarchies.

As already observed, some of the Assumptions A1-A8 are not typically satisfied by commercial machines, where data movement operated by various virtual storage mechanisms do not comply with literal execution (A3). Nevertheless, often, the departure from literal execution is not sufficient

to elude the bottlenecks captured by the lower bound arguments in the proof of Theorem 2. For example, cache systems typically move data to addresses with lower latency only in response to an instruction that misses. Therefore, the argument leading to Bound 17 on T_1 applies. Before data are requested from slower levels of the hierarchy, room must be reserved in the faster one. If no room is available, then the memory instruction causing the miss cannot be submitted. Thus, the principle of non-overlapping flight essentially applies and so does the argument leading to the bounds on T_{2j}. We ought to realize that, due to evictions, a move per cache level could take place on a single miss. Since the number of levels could reasonably grow logarithmically with N, one needs to be careful when relating number of moves to number of instructions. This issue is taken care of by the fact that, in the proof of Theorem 2, only moves of the elements in I' toward locations with smaller latency really need to be counted. Technically, the bounds for both T_1 and T_{2j}'s must be divided by the cache line length. Thus, constant length must be assumed, to ensure that the asymptotic nature of the bounds is not affected. Similar, and in fact simpler, arguments can be developed for register renaming in modern processors.

Ultimately, we are interested in machines that can actually achieve better performance than those constrained by Assumptions A1-A8. This will be the focus of the next section.

5 Processor Organizations

In this section, we turn our attention to the processor side of the machine. To set the stage, we first discuss a straightforward organization, called the canonical processor. We then develop two novel processor organizations, respectively called the Speculative Prefetcher and the Speculative Prefetcher and Evaluator, which will yield the first machines capable of executing all direct-flow programs with constant CPI, thus going beyond what can be achieved by EDM machines. The main performance metric we target in the processor design is the execution time of programs. We also consider the "cost" of a processor, in terms of the space occupied by its layout. For time and space estimates, we assume the availability of a functional unit $\mathcal{F}$, that can be laid out in constant space and that can apply any operator included in the instruction set, in constant time.

5.1 The Canonical Processor

In this subsection, we consider a simple organization, called the *canonical processor*, which executes programs one instruction at a time, with no memory concurrency among accesses made by different instructions. For such a processor, the main avenue for performance optimization consists in exploiting locality of references, to reduce the latency of individual memory accesses. The performance of the canonical processor provides a baseline against which we can measure other processors, like the one proposed in the next section, where performance is pursued also by exploiting concurrency of memory accesses. The canonical processor is easily realized in constant space, regardless of the size M and P of data and program memory, respectively.

Consider at first the execution of just one instruction I, say of data-processing type: $I = \langle op,\ (\hat{x},\ x),\ (\hat{y},\ y),\ (\hat{z},\ z)\rangle$, using direct or indirect addressing modes. Execution completes when the processor submits the store request for its destination. Since the memory and the processor operate concurrently, the time for actual storing of data in the memory is not relevant here. Observing that the loads for the two source operands and the load for an indirect destination address (if present) can be submitted one after the other, so that the corresponding fetches are performed

concurrently, the time bound for the execution I when the data memory is in state m, is given by

$$\begin{aligned} T(I,m) &= \max\{a(x)+\hat{x}*a(m[x]),\ a(y)+\hat{y}*a(m[y]),\ \hat{z}*a(m[z])\}+O(1) && (20)\\ &= O(a(\mu)), && (21)\end{aligned}$$

where the constant additive term $O(1)$ accounts for the time steps required to submit the load/store commands to the memory one at a time and the constant time required to perform the operation. Here, μ denotes the maximum address of a location from which instruction I is loading, whether directly or indirectly. Observe that the dependence upon m is effective only when operands are read indirectly.

Consider now a program execution resulting in the instruction stream $\mathcal{I} = (I_1, I_2, \ldots I_N)$, where $I_j = \langle op_j,\ (\hat{x}_j,\ x_j),\ (\hat{y}_j,\ y_j),\ (\hat{z}_j,\ z_j)\rangle$. The canonical processor will execute the stream sequentially, taking memory through the sequence of states, $m_0, m_1, m_2, \ldots, m_N$, producing m_j from m_{j-1} by executing I_j. The total execution time is then

$$T_{can}(\mathcal{I}, m_0) = \sum_{j=1}^{N} T(I_j, m_{j-1}) = O(\sum_{j=1}^{N} a(\mu_j)). \tag{22}$$

If all addresses accessed by the program are smaller than μ, then $T_{can}(\mathcal{I}, m_0) = O(Na(\mu))$. In terms of the CPI metric, we can rewrite Equation (22) as

$$CPI_{can}(\mathcal{I}, m_0) = O(\frac{1}{N}\sum_{j=1}^{N} a(\mu_j)). \tag{23}$$

Thus, the CPI is proportional to the average access latency over all the program, reiterating that performance improvements are tied to locality of references (low μ_j's). To conclude this subsection, we observe that, as a model of computation, the canonical processor with hierarchical memory closely resembles the Hierarchical Memory Model (HMM) of [2].

5.2 Machines with Speculative Prefetching

We now introduce an approach based on speculative prefetching of memory values into the processor. Within this approach, we will develop two specific processor designs, the *Speculative Prefetcher* (SP) and the *Speculative Prefetcher and Evaluator* (SPE). The present subsection presents the features that are common to the SP and the SPE processors. Subsequent subsections develop the features that distinguish the two designs.

5.2.1 Overview of the Organization

Figure 6 shows the top level structure of a machine organization aimed at providing an efficient, extensible physical realization of the ideal RAM(M, P). At this level, the structure is common to both the SP and the SPE processors. The main components are an *Instruction Generator Unit*, IGU(n), connected to a pipelined hierarchical program memory of size P and access function $a(x)$, denoted by $a(x)$-PHM(P), and an *Instruction Execution Unit*, IEU(n), connected to a pipelined hierarchical data memory of size M and access function $a(x)$, denoted as $a(x)$-PHM(M). Parameter, n, called the *processor size*, determines the resources present in the IGU and in the IEU, as described

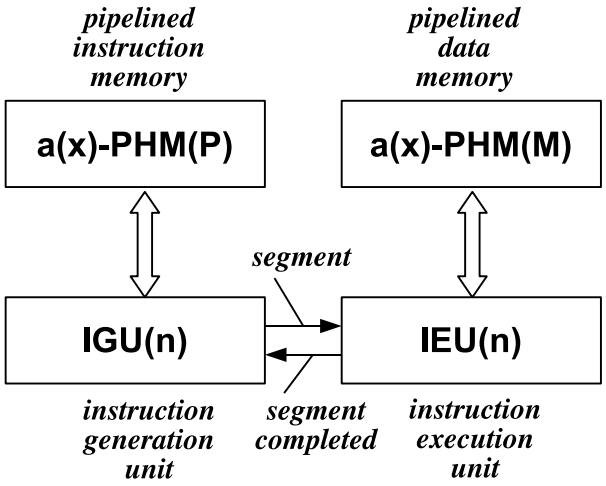


Figure 6: Top level organization of machines with speculative prefetching. Machine parameters P, M, and n respectively denote program-memory, data-memory, and processor size. The latency to access address x is $a(x)$. Segments of at most n instructions are composed by the IGU and executed by the IEU.

later. The value of n will typically be chosen to be proportional to the maximum latency $a(M)$ of the data memory.

The execution of an arbitrary program is organized as a sequence of *stages*. In each stage, a *segment* of the dynamic stream of instructions is processed. Each segment is restricted to have at most n instructions. The general objectives of the processor design are (i) to overlap memory accesses of different instructions within the same segment and (ii) to manage the flow of information between instructions in the same segment within the processor itself, thus eliminating some memory accesses.

In each stage, the IGU forms the next segment and feeds it to the IEU for processing. The operation of IGU is illustrated in Figure 7. Given an initial value for the program counter, pc^1, the IGU reads instructions starting at that address, and forms segment 1, consisting of instructions $G^1_i, i = 1, 2, \ldots$, where the superscript indicates segment number and the subscript indicates instruction number. When it reads a branch instruction, the IEU obtains the predicted target address of the branch, using some prediction mechanism, and continues to fetch instructions from that address. In the illustration of Figure 7, the instruction $G^1_{r^1}$ is a branch and the next instruction, G_{r^1+1}, is from its predicted target address. This continues until the desired number, q^1, of instructions are composed into the segment. The segment is then passed to IEU, which executes the sequence of instructions in the segment and terminates it when it reaches the end or encounters a mispredicted branch. In the illustration, the IEU terminates segment 1 after it determines that the branch, $G^1_{r^1}$, is mispredicted and returns indicating that the real target after instruction r^1 is pc^2. Thereupon, the IGU reads instructions from that address and forms segment 2 of desired length q^2. The figure illustrates complete execution of segment 2 till the end, as all branches are correctly predicted and the IEU indicates this. In the next step, the IGU computes the third segment, from where it left off at the end of segment 2 and this process continues. At the end, as illustrated in

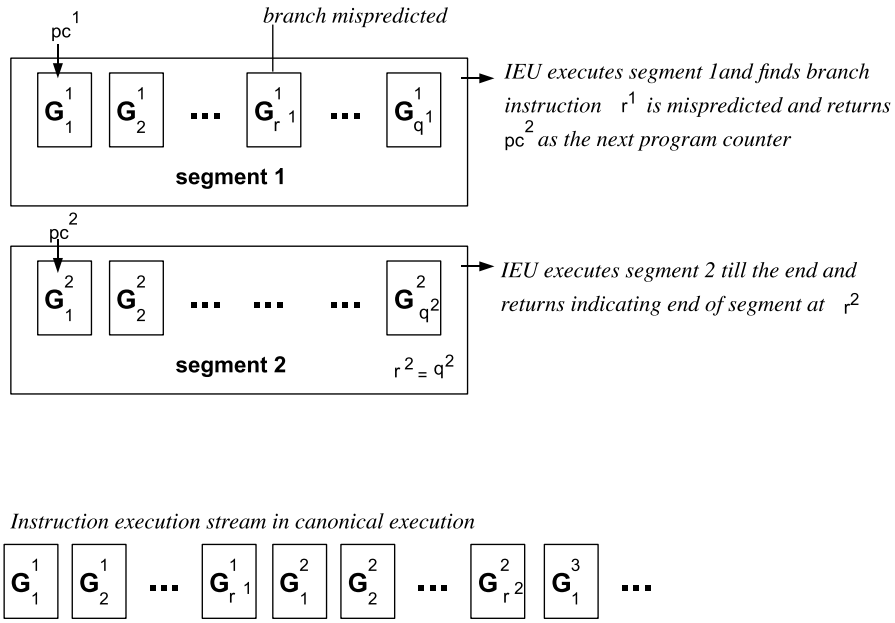
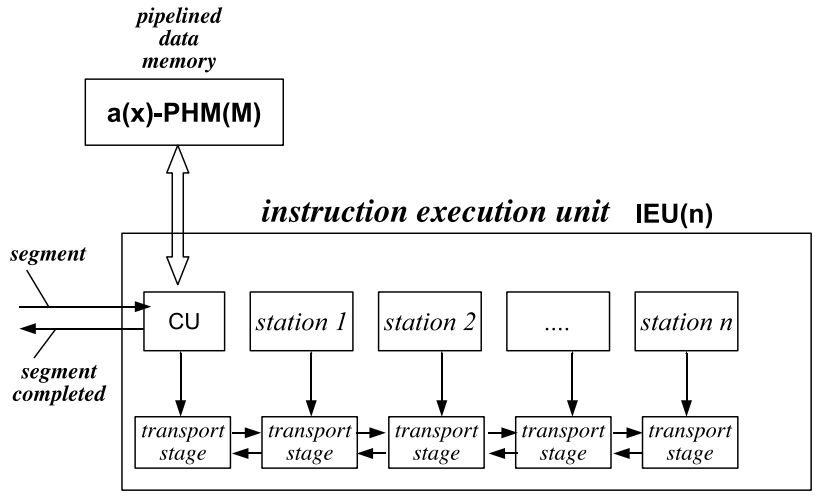


Figure 7: Instruction segments generated by IGU(n) and executed by IEU(n). For segment s, $q^s \leq n$ instructions are generated and $r^s \leq q^s$ are committed.

Figure 7, the instruction sequence executed, $G_i^1, i = 1 \ldots r^1, G_i^2, i = 1 \ldots r^2, \ldots$ corresponds to the sequence in canonical execution. Specifically, if $k = r^1 + r^2 + \ldots r^{s-1} + i$, with $i < r^s$, the $G_i^s = I_k$. The degree of freedom IGU has in choosing the lengths, $q^1, q^2, \ldots$ for the segments (rather than choosing the maximum n, each time) can be helpful, as illustrated in later sections.

The general structure of an IEU is depicted in Figure 8. It consists of a control unit (CU) and n stations. The CU interfaces with the data memory and coordinates all requests submitted to the memory and all responses coming out of memory. It also controls information flow into the stations. Each station is capable of storing a *template* of an instruction from the program segment. A template contains a number of "program" fields and a number of "auxiliary" fields. Program fields include opcode, operand addresses, addressing modes, and instruction address (for verification of branch speculation). Auxiliary fields include values of operands and result, as well as of addresses in the case of indirect mode, and the identification (number within the segment) of instructions that fetched or created source values for this instruction (to properly manage values based on speculation). Each field has indicators to specify whether the field is valid/invalid/expected etc. Auxiliary fields are updated as part of the execution process. The CU will also orchestrate the appropriate shift of the instruction templates between the stations. The CU is equipped with a functional unit. Optionally each station may also contain a functional unit.

The interconnections between the CU and the stations of processor are design-specific. Figure 8 shows some common communication operations used. Two commands describe the desired communication patterns: The *Send* command can be used to send a message between two agents, where each agent can be either CU or some station. Starting from the source, the message hops one intermediate station per time step and reaches the destination. Only the receiver will read the content and act on it. The *Forward* command is used by a source agent to send a message to all



Transport stages can transfer a constant number of messages between neighboring stations (or CU and station 1) in each direction in one time step.

CU and stations can communicate using the following primitives which are implemented by the collection of transport stages:

send(dest, msg) : the message is sent to the specified destination, which can be the CU or any station
forward(msg): the message is forwarded to all stations to the right of the sender the sender can be either CU or any other station

Figure 8: Structure of the instruction execution unit IEU(n).

stations to its right. The message being forwarded moves unchanged by one station per cycle and is read by all stations it traverses.

5.2.2 Strategy for Instruction Execution

When a segment is submitted to the IEU, it is processed in two non-overlapping phases: the *initialization phase* and the *execution phase*. The initialization phase will be common to the SP and the SPE processors, thus it will be completely described here. The execution phases of the SP and the SPE are different, thus their details will be developed in subsequent subsections.

Without loss of generality, we describe the processing of a segment, of size q, consisting of the sequence of instructions, $G_1 \dots G_q$, which is submitted to the IEU when the state of memory is m_0 and the execution commits the initial subsequence $G_1 \dots G_r$, where G_r is the first mispredicted branch in the segment, if any, or $r = q$.

Initialization phase. In the initialization phase, the control unit initializes the stations with templates of instructions from the segment, in program order, using the *send* command. That is, for $j = 1, \dots, q$, station j will hold the template for instruction G_j. A speculative version of the instruction operands is also obtained from memory and sent to the corresponding station. That is, for each operand, the value from the initial memory state m_0 is obtained and stored in the

corresponding template. This is accomplished in three sub-phases, executed one after the other, as described below:

Sub-phase 1: reading the instructions. For $j = 1, 2, \ldots, q$, the CU reads each instruction G_j in the segment, composes the template for G_j with the program fields initialized, sends this template to the j^{th} station, and submits to the memory a (*speculative*) load request for the locations corresponding to the direct addresses in all source operands and the direct address of an indirect destination operand (if present). Each load request is tagged with information identifying the instruction and the addressing mode. Since a constant number of time steps are spent for each instruction, this sub-phase takes time (the second subscript indicates subphase number)

$$T_{init,1} = O(q).$$

Sub-phase 2: reading directly-addressed locations. The CU extracts from the memory response queue the result of each load request of sub-phase 1, which contains an address x, the loaded value $m_0[x]$, and the tag. If the tag indicates the need for indirection, another load request is submitted to memory, for address $m_0[x]$, this time with a tag indicating direct mode. A packet with x and $m_0[x]$ is sent to the station j holding the instruction G_j (j being specified by the tag) for which the load was performed, so that the appropriate updates can take place. Since the latencies for directly-addressed locations are overlapped, this sub-phase takes time

$$T_{init,2} = O(q + \max_{j=1}^{q}\{a(x_j),\ a(y_j),\ \hat{z}_j * a(z_j)\}),$$

where, strictly speaking, the notation refers to data-processing instructions, but can be naturally adapted to branch instructions as well. To see the role of term $\hat{z}_j * a(z_j)$, recall that $\hat{z}_j = 0$ when the result is written direclty and $\hat{z}_j = 1$ when the result is written indirectly. Clearly, during initialization, the value of $m_0[z_j]$ is (speculatively) loaded only in the latter case.

Sub-phase 3: reading indirectly-addressed locations. The CU extracts from the memory response queue the result of each load request of sub-phase 2 and sends the corresponding $(m_0[x], m_0[m_0[x]])$ pair to the appropriate station. Since only indirectly addressed operands are involved, this sub-phase takes time

$$T_{init,3} = O(q + \max_{j=1}^{q}\{\hat{x}_j * a(\tilde{x}_j),\ \hat{y}_j * a(\tilde{y}_j)\}),$$

where, as defined in Section 2.1, $\tilde{x}$ denotes the actual data address.

At the end of this phase, all the templates are initialized with operand values from initial state of the memory. If all addresses accessed by the segment are smaller than M_{init}, then the total time for the initialization phase can be bounded as

$$T_{init} = T_{init,1} + T_{init,2} + T_{init,3} = O(q + a(M_{init})).$$

Execution phase. In the execution phase, the instructions placed in the stations are executed and, at the end, the resulting memory state (consistent with canonical execution) is updated. Many variants are possible in accomplishing this task, depending upon the extent to which the processor indulges in speculation. The greater the amount of speculation, the greater will be the performance benefit when speculations are correct and the greater will be the performance loss

when the speculations are incorrect. This tradeoff will be illustrated by contrasting the SP and the SPE designs.

The primary exploration here is to speculate on the assumption that the initial memory state obtained in the initialization phase is sufficient to process all the instructions from the segment, without further access to the memory. Furthermore, if the results of store operations performed in the segment are *directly* used by other instructions in the same segment (functional dependence), memory accesses to these locations can be avoided by internally forwarding the results within the stations. Such speculation fails only when an instruction stores into a location, x, which is used as an indirect operand ($\hat{x} = 1, x$) by some later instruction in the segment (address dependence). The SP and the SPE differ in how they deal with such failures, due to address dependence. When the speculated operands of an instruction are detected to be invalid, the latest contents of the desired memory locations are obtained and the instruction is correctly executed. However, the result of this instruction might render the speculation made by some succeeding instruction as invalid. Thus, in general, an instruction might speculate more than once in its life time. The SP does not permit an instruction to speculate more than once, whereas the SPE entertains instructions to speculate many times. As mentioned before, the initial phases are identical for these two designs. Their execution phases are different and are discussed separately, in Sections 5.3 and 5.4.

5.3 The Speculative Prefetcher

We now focus on the specific features of the SP, the simpler of the two processors.

5.3.1 The Execution Phase of the SP

In the SP, the execution phase takes place in a sequence of *rounds*, until the segment ends or a branch is detected to have a mispredicted target. In each round, one instruction (in program order) is committed, *i.e.*, its result is stored into memory. With reference to a segment $G_1, \ldots, G_q$ of size q in which the first r instructions are committed ($r = q$ or G_r is a mispredicted branch), exactly r rounds will be executed. At the beginning of each round $j = 1, \ldots, r$, each station $i = 1, \ldots, q - j + 1$ has instruction G_{j+i-1} in it. Only station 1 actually performs the operation of its instruction G_j, while all the other stations update any latest information (forwarded by preceding stations) regarding their operands. At the end of the round, station 1 commits G_j and all stations shift their instruction templates once to the left. At this point, the data memory reflects the correct state m_j of the canonical execution. In round $j = 1, \ldots, r$, the CU orchestrates the following sequence of steps, called *sub-rounds*.

1. *Sub-round 1: operand acquisition.* The CU extracts from the memory queue (*cf.* Section 3.5) and forwards to all stations any response packets (to read requests submitted to memory in previous rounds) until all the operand values for G_j are received. Note that, for $j = 1$, the operands for G_1 are guaranteed to be ready after the initialization phase. For $j > 1$, consider the case when G_j indirectly reads from a location $m[x]$ and some prior instruction G_i ($i < j$) modifies the direct address x ($v_i = x$ in sub-round 4 of round i). A read request of $m[x]$ will be submitted in sub-round 6 of round i and all subsequent instructions learn about this circumstance by the forwarding occurring in sub-round 5 of round i. If the read request is still pending at the beginning of round j, then this will be reflected in the template for G_j providing the CU with the necessary information to implement sub-round 1, as described above.

2. *Sub-round 2: branch handling.* If G_j is a branch, then the computed target is compared by the CU with the speculated target. If the two are equal (correct prediction), then sub-rounds 3-6 are skipped, as they are needed only for data-processing instructions, and sub-round 7 is executed. Else (misprediction), the segment is terminated: the CU records the computed target (which will provide the *pc* value for the next segment) and sends a termination signal to station 1.
3. *Sub-round 3: computation of results.* The functional unit computes the result value, v_j, of instruction G_j.
4. *Sub-round 4: write back.* The CU sends to the memory a request to store v_j into the location $\tilde{z}_j$ specified by the instruction G_j.
5. *Sub-round 5: forwarding of results.* The CU forwards to all stations ($i > 1$) a packet with the information that value v_j is being stored in location $\tilde{z}_j$ by instruction G_j.
6. *Sub-round 6: eager prefetching.* The CU submits an *eager* load request to memory location v_j, tagged with the instruction number j (for the possible benefit of any subsequent instructions with an indirect operand through location $\tilde{z}_j$).
7. *Sub-round 7: instruction shifting.* The CU requests all stations to shift their template to thier left.

In each round, $j = 1 \ldots r$, each station $i > 1$ takes the following actions in each time step. Initially only station q is marked as the last station.

1. *Termination.* If a termination signal is received from station $i - 1$, then if this is not the last station, then termination signal is sent to station $i + 1$. This station then marks itself as inactive.
2. *Operand acquisition.* If a data value either the result of an instruction (forwarded in sub-round 5) or the result of a memory request (eagerly submitted in sub-round 6) is received by this station, then the address and the originating instruction number are examined. If it is a revised data value for any of the operands of the instruction template currently at this station, then the operand information is updated accordingly. Otherwise, there is no action.
3. *Instruction shifting.* If station $i - 1$ requested a template shift, then the contents of the current template are sent to station $i - 1$ along with the indication whether this is currently the last station. If this is the last station, it marks itself as inactive until initialized again. If this is not the last station, then in the very next cycle, it must first receive the template from station $i + 1$ and overwrite its template with those contents. It marks itself as the last station when the received template indicates so.

By and large, the implementation of the above sub-rounds is straightforward. However, the full specification of the details can be cumbersome. Here, we underline only a few main points.

- Clearly, the memory responses to the read requests generated in sub-round 6 arrive somewhat asynchronously with respect to the round sequence. The CU simply forwards these responses to all stations. If the location being read is relevant to an instruction, the corresponding template will be updated accordingly. The auxiliary fields in a template can be suitably designed to do this operation very efficiently.
- Some subtleties are involved with the implementation of the shift of the instruction templates involved in sub-round 7. A naive solution would require time proportional to the number of templates to be shifted. The solution indicated is based on the systolic paradigm and can be

very efficient. Essentially, the shift operation is pipelined across the stations so that the CU can operate at a constant rate, even though the commands will reach the stations only after suitable delays. Some care needs to be exercised in managing the interaction between the upstream and the downstream flow of information across the stations. The ideas needed are very similar to those introduced by [27] for systolic implementations of priority queues and will not be described here.

- Another implementation subtlety arises when, due to a mispredicted branch, sub-phase 2.1 terminates, but some of the eager memory requests (sub-round 6) are still outstanding and will complete at a later stage. Waiting for all these requests to be satisfied before beginning with the next stage could result in inefficiencies. A simple solution is to include the stage number in the memory requests, so that the answer can be simply discarded by the CU, if it arrives at a later stage.

We observe how direct inputs of an instruction modified by a previous instruction in the segment are updated by the forwarding mechanism, with no access to memory. Thus, for each round, some time is spent waiting for the indirect operands (sub-round 1) plus a constant time for the remaining sub-rounds (2-7). Given the preceding considerations, if all addresses *indirectly* accessed by the segment are smaller than $M_{indirect}$, then a straightforward bound on the time taken by station to commit r instructions is

$$T_{exec,1} = O(r * a(M_{indirect})).$$

After r instructions are committed, the data memory is in state m_r. Furthermore, station 1 holds the last committed instruction of the segment, G_r, which provides the appropriate value for the program counter from which the next segment starts. If $r < q$, the terminate signals must still travel to the last station before all stations become inactive. This propagation takes time

$$T_{exec,2} = O(q - r),$$

so that the total time for the entire execution phase is

$$T_{exec} = T_{exec,1} + T_{exec,2} = O(r * a(M_{indirect}) + q).$$

We can observe how the propagation of termination signal could be easily overlapped with the subsequent stage; the corresponding performance improvement would be limited to constant factors.

5.3.2 Performance Considerations

Combining now the bounds for the initialization and the execution phases of the segment (with r instructions committed out of q instructions), we can bound the time T for the entire segment as

$$T = T_{init} + T_{exec} = O(q + a(M_{init}) + ra(M_{indirect})).$$

Finally, considering the whole program having S segments, with each segment $s = 1, \ldots, S$ having the corresponding parameters, $r^s, q^s, M^s_{init}, M^s_{indirect}$, the total execution time for the program can be written as sum of the times of all segments:

$$T = \sum_{s=1}^{S} T^s = O(\sum_{s=1}^{S}(q^s + a(M^s_{init}) + r^s * a(M^s_{indirect}))). \tag{24}$$

The preceding bound could be refined by a more accurate analysis. For example, the factor r^s could be lowered if many instructions do not perform indirect accesses. One could also estimate more precisely accesses occurring at addresses smaller than $M^s_{indirect}$ and the overlap of access latencies resulting from eager prefetching (sub-round 6). Indeed, in [11], an analysis approach is developed whereby a suitable *latency graph* is associated with the execution of a program on the SP processor and it is shown that the SP execution time is essentially the weight of the heaviest path in such graph.

Next, we provide some intuition on which properties of a computation are captured by the terms in the right hand side of the above relation.

- *Locality of references* is partially captured by the terms M^s_{init} and $M^s_{indirect}$, which bound the region of memory accessed during the first and the second phase of the stage, respectively.

- *Branch prediction accuracy* is reflected by q^s. As the bound shows, values of q^s larger than r^s, the number of instructions actually committed, increase the time of the segment without increasing its productivity. As an indirect effect, M^s_{init} can also increase with q^s, as later instructions may access higher addresses.

- *Address dependence* also has an important impact on performance, as indicated by the last term of Equation (24). For our current purposes, it suffices to observe that if instruction G_j is address-dependent upon instruction G_i, then some data for G_j must be fetched again from the memory, after G_i has executed. This memory latency translates into an unavoidable delay between the executions of G_i and G_j.

In general, the SP can take advantage of both the pipelined nature and the hierarchical nature of the memory to which it is connected. For direct-flow programs, the pipeline capability turns out to be sufficient to obtain the following simple, nevertheless non trivial, result.

Proposition 6 *An SP with a pipelined memory with latency at most $a(M)$ can execute any* direct-flow *program of N instructions, accessing memory locations with address smaller than M, in time $T = O(N + a(M))$, using $n = a(M)$ execution stations.*

Proof: The instruction stream can be partitioned into $S = \lceil N/n \rceil$ segments with sizes $q^1, \ldots, q^{S-1} = n$ and $q^S = N \bmod n$. Since there are no branches, all the instructions generated in a segment are completed in that segment ($r^s = q^s$). Sine there are no indirect addresses, $M^s_{indirect} = 0$. By assumption, $M^s_{init} \leq M$, for any s. Then, straightforward manipulations of Equation (24) yield the stated result. □

We observe that, for $N \geq a(M)$, the execution time in Proposition 6 becomes $T = O(N)$ and correspondingly $CPI = O(1)$. In contrast, for the canonical processor, $T_{can} = O(Na(M))$. To within a constant factor, the memory latency is successfully hidden by the SP, for direct-flow programs. That a processor ought to be able to process instructions at a constant rate if all addresses are known in advance and the memory is pipelinable may deceivingly appear as obvious. In order to appreciate Proposition 6, its result ought to be contrasted with the superlinear lower bounds obtained in Section 4 for EDM machines, which include several previously proposed models of computation with pipelined memory.

The SP is not an EDM machine. Specifically, it violates the read-after-write assumption A6 of Section 4.1, which disallows overlapping execution of an instruction I_h writing a location x with a later instruction I_k reading from it. In fact, in the SP, I_h and I_k will be concurrently in flight if they

fall in the same segment, as both instructions will be issued in the initialization phase, before any of them commits. While, in this phase, I_k will fetch the incorrect operand value from x, the correct result of I_h will be later forwarded to I_k inside the IEU, thus preserving the program semantics.

It is interesting to note that, beside A6, the SP does not violate any other EDM assumption, as one can easily check. In particular, A3 is technically satisfied: since the SP does fetch the (possibly wrong) contents of x directly from the memory, flight time is at least $a(x)$. As for Assumptions A7 and A8, the corresponding parameters are $s = n$ and $f = n$, where n is the number of stations of the SP. We may also observe that the upper bound to ALU concurrency arising for EDM machines from the c-coverability of the computation (*cf.* Proposition 5) is based on assumption A6, hence it does not apply to the SP.

To fully appraise the potential of the SP, we stress that the SP executes *any* direct-flow program in time proportional to its number of instructions, without any programmer or compiler effort. In contrast, in previous EDM models like BT and PH-RAM, most direct-flow programs are quite inefficient. Generally, faster equivalent programs can be obtained by clever restructuring, involving the insertion of suitable copy/move instructions. Even then, their performance may fall short of what is achieved by the SP.

The capabilities of the SP processor go well beyond direct-flow programs. For example, it is shown in [11] that quicksort can be implemented on the SP, with running time $T = O(C)$, where C is the number of comparisons executed by the algorithm. Clearly, since at least C instructions must be executed, we have that $CPI = O(1)$. This result is interesting in view of the fact that the quicksort algorithm has a significant dependence upon the input data. Indeed, the very number of instructions can vary considerably, from $\Omega(K \log K)$ to $O(K^2)$, across inputs sequences of a given size K. At the program level, this variability can be traced to a richness in branches and in address dependences. However, this result is achieved at the expense of considerable complication, including an undesirable amount of static loop unrolling, which drastically increases program size. In the next subsection, we discuss this drawback of the SP processor on a simple example. In Section 6.2, we will present a simple, optimal quicksort implementation for the SPE processor, which circumvents the limitations of the SP.

5.3.3 Limitations of the SP and Motivation for the SPE

For programs that are not direct-flow, address dependences pose some performance problems on the SP. An address dependence relation, $j \xrightarrow{*w} k$, between a pair of instructions j and k forces the latency of access to location w to occur within the interval between completing instruction j and fetching the correct operand for instruction k. If i and j occur next to each other in a segment, the SP will not issue any additional memory requests in that interval and the only possibility for overlap would be any memory requests that may have been issued earlier. Adjacent occurrence of j and k within a segment could manifest itself in many natural ways of programming, as illustrated by the following simple example that increments all the elements of an array. Clearly accesses to different elements of the array are independent of each other and one would expect their latencies to be overlapped. A naive program is of the form: `for` *i=1* `to` *n; A[i] = A[i]+1.* In the SP, using good branch prediction, the IGU might be able to roll out a segment of instructions in the following form, where the pseudo-code assumes that a is initialized with the beginning address of array A: $a = a + 1;\ m[m[a]] = m[m[a]] + 1;\ a = a + 1;\ m[m[a]] = m[m[a]] + 1;\ a = a + 1;\ m[m[a]] = m[m[a]] + 1 \ldots$. Notice the address dependence between the first two instructions, then between the next two instructions and so on. But, there is only functional dependence between different instances of instruction

$a = a + 1$. Hence, potentially, all the addresses can be generated without further memory accesses and therefore all the elements of the array should be accessible concurrently. However, when executed on the SP, in the initialization phase, all instructions will get the initial contents of a, which is the address of the first element of A. Only after each update to a can the correct value of the corresponding element of the array be loaded from memory. Thus, no concurrency is exploited among the accesses to different array entries.

The problem just described can be circumvented if the loop is actually unrolled, for example by a compiler, with the value of a for each iteration directly embedded in the instructions. However, this solution is not ideal, since the amount of unrolling increases with the problem size. More importantly, precomputing the addresses statically is not feasible if further levels of indirections are involved, as in the example `for` *i=1* `to` *n; A[i] = A[B[i]]+1*.

In view of the preceding observations, it would be profitable to extend the SP in two ways. In SP, in each round, only the instruction in station 1 executes and submits new fetch requests to memory.

1. In each round, we will now permit all stations to execute speculatively with whatever information they have at that time and also provide them an opportunity to submit fetch requests to memory at the end of a round.
2. Furthermore, in each round, we will permit each station to receive the latest result information from its preceding stations, so that its speculative execution will be based on latest information available at that time.

The first provision will enable the processor to execute all independent instructions in a segment concurrently within one round. The second provision will make such execution up-to-date with respect to latest available information and will permit all instructions along any functional dependence chain within the segment to be executed in one round. For instance, in the array incrementing example discussed above, all the instances of the instruction $a = a+1$ can be executed in one round and fetches for all the correct elements of the array can be issued at the end of that round, overlapping their latencies. A disadvantage of the second provision is that a round, now, takes q steps to execute, where q is the size of the segment. The resulting machine is the SPE, described next.

5.4 The Speculative Prefetcher and Evaluator

The organization of the SPE is very similar to that of the SP, except for a richer set of states implemented at the stations. For simplicity, we also assume that each station is equipped with its own functional unit.** As in the SP, the execution of a segment takes place in the SPE as a sequence of two phases: an initialization phase followed by an execution phase. The initialization phase is identical to that of the SP. Given a segment consisting of instructions, $G_1, \ldots, G_q$, the IEU loads q stations with the templates of the q instructions, issues memory fetches to all direct and indirect operands from the current state of memory, forwards the results coming from the memory to all the stations, so that the template for each station is initialized with all speculative data from the current state of memory. The execution phase of SPE differs substantially from that of the SP and is described in the next subsection.

**In fact, one functional unit in the entire processor would suffice to achieve the same time bounds to within constant factors, at the price of some complication in the description of the execution process. Ultimately, the ratio between the number of stations and the number of functional units would be dictated by engineering considerations.

5.4.1 Execution Phase of the SPE

The execution phase consists of a sequence of one or more *rounds*, which are substantially different from those of the SP. In each round, one or more consecutive instructions of the given segment $G_1, \ldots, G_q$ are completely executed and their results are committed to memory. Some of the remaining instructions are speculatively executed: correct speculation generally helps performance, by reducing the number of rounds R. The latter is in the range $1 \leq R \leq q$, with the exact value depending both on the instructions in the segment and on the memory state just before the segment is executed.

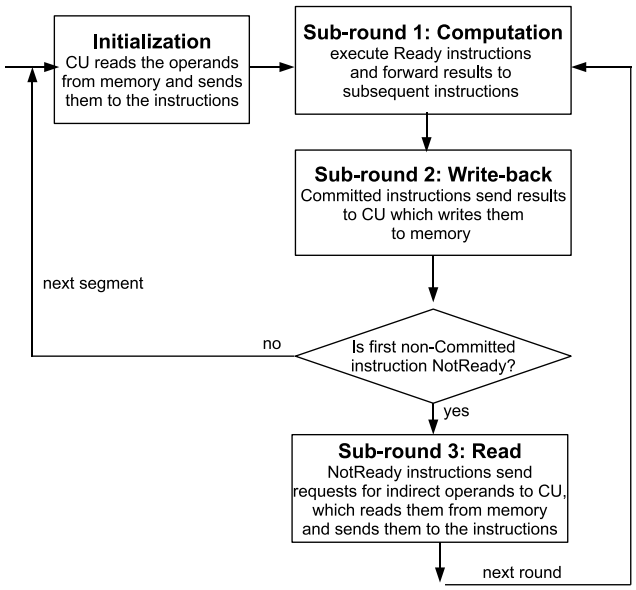


Figure 9: Repeated execution of subrounds in the SPE execution phase.

At the beginning of each round, as a result of either the initialization phase (for the first round) or of the previous round, templates for the q' instructions not committed during the previous rounds are held by the first q' stations of the IEU, with all the operands suitably initialized. Each round consists of a sequence of three sub-rounds as depicted in the flow chart of Figure 9.

Sub-round 1: Computation. In the first sub-round, both the order of activation of the stations and the direction of the information flow is from left to right. The general structure is as follows where, to be specific, we say that a message is received (rsp., sent or forwarded) by station i at time t if that message is an input (rsp., output) for the state transition of station i at time t. At time 0, the CU generates and forwards a start signal which is received by station i at time i, for $i = 1, \ldots, q'$. During time steps $i, \ldots, 2i-2$, station $i > 1$ (a) receives the messages respectively generated by stations $1, \ldots, i-1$ (as part of their forward transition described below) and (b) undergoes a receive transition (see below). Each station i undergoes a forward transition just once, at time $2i$, the resulting state being its final state for that round. As an output of this transition, for $i < q'$, a message is forwarded, which is received by station $i+k$ at time step $2i-1+k$. The last activity of the sub-round occurs at time $2q'$, at station q'. Hence the time for this sub-round can be bounded as $t_{compute} = 2q' = O(q)$. The states and the transitions are now explained.

Each station can be in one of four *states* (see Figure 10) with the following names and meaning. *Ready*: the station has all operand data (possibly speculative) for its instruction. *NotReady*: some indirect operand value needs to be read from memory. *Committed*: the result of the instruction is final and can be committed. *Flushed*: the instruction will not be executed any more, due to a branch misprediction earlier in the segment. At the beginning of a round, all states are initialized to be Ready.

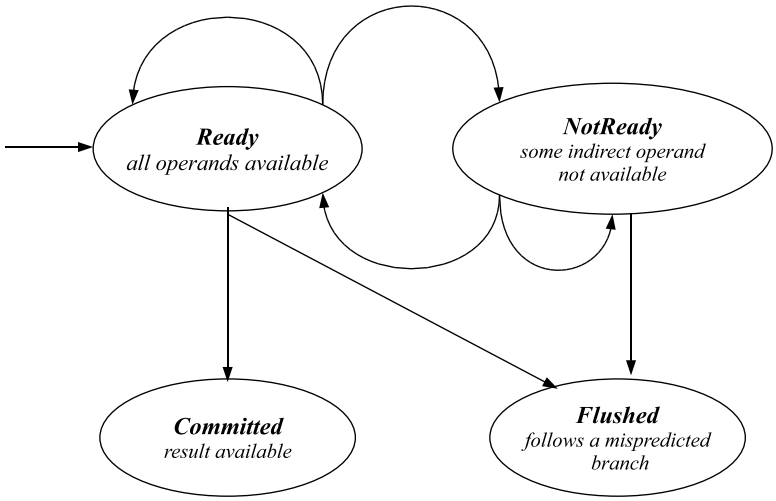


Figure 10: State transitions of stations that can occur during the computation sub-round of the SPE execution phase.

The stations undergo two types of transitions, as illustrated in Figure 11. Whenever a station receives a message, it examines the message and undergoes a state change as explained under *receive transitions* below. If the transition was a result of a message from the immediate predecessor to its left, in the next time step it makes another state change described under *forward transitions* below and then forwards a message to its successor stations. The dynamics of state changes and message flow is illustrated for a 4-stage pipeline in Figure 11.

Receive transitions: A message carries the forwarding station number, the current state of the sender and the address and value of the new result produced by the sender (if any). When station i receives a message from one of its predecessors, it examines if the result implies a change to its operands: if so, it updates the appropriate operand values. If the updated data is an indirect operand address, the station changes its state to NotReady (as that indirect operand must be obtained from memory). Otherwise, it remains in the Ready state.

Forward transitions: After station i updates its state as a result of a message from its immediate predecessor, it takes the following actions:

1. If its state is Ready, it executes the instruction and records the result value in its template.
2. If predecessor's state is Flushed, then it sets its state to be Flushed.
3. If predecessor's state is Committed and its state is Ready, then it sets its state to be Committed.
4. Finally it forwards a message to its successors giving its state and result. The only exception is that if its state is Committed and it detected a branch mispredict in its execution, it

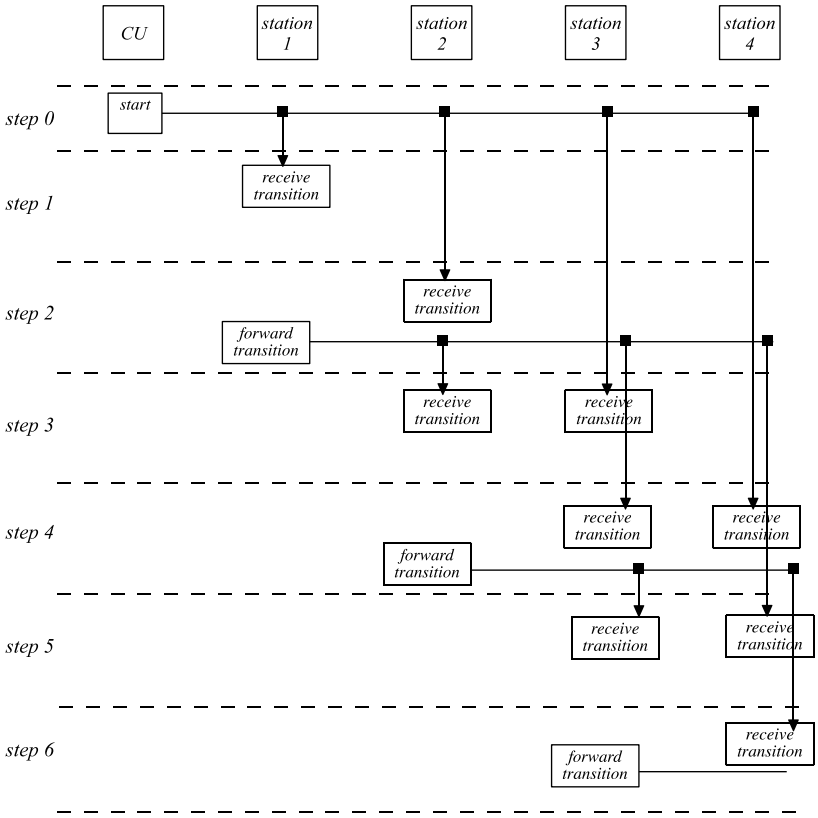


Figure 11: Illustration of message flow causing state transitions of stations during the computation sub-round of the SPE execution phase.

communicates a Flushed state to its successors.

From the preceding transition rules, it is an easy exercise to establish that, at end of the compute sub-round, the general situation is as follows. For some c with $1 \leq c \leq q'$, stations $1, \ldots, c$ are in the Committed state. Either all or none of the remaining stations are in the Flushed state. In the former case (stations $c+1, \ldots, q'$ Flushed) or if $c = q'$, all instructions have been completely processed and the current round will be the last round. Otherwise, among stations $c+1, \ldots, q'$, some will be NotReady (in particular, station $c+1$) and some may be Ready. With this premise, we describe the remaining sub-rounds.

Sub-round 2: Write-back. The main goal of this sub-round is to update the memory state to reflect the Committed instructions and to read the missing operands for the NotReady instructions. This is accomplished by having each station send to the CU a packet, with the station number and its current state. In addition, if the state is Committed, the packet contains the address and data to be stored. If the state is NotReady, the packet contains the operand addresses whose contents are needed. Upon receiving a packet for a Committed instruction, the CU will submit to the memory the appropriate write request. If all packets are Committed or if the first non Committed packet is Flushed, then the current round as well as the entire execution phase terminates. Otherwise, sub-round 3 will complete the current round and then another round must be started.

It is straightforward to orchestrate sub-round 2 activities in a pipelined manner, each station taking constant time, so that the time for the sub-round can be bounded as $t_{write} = O(q') \leq O(q)$.

Sub-round 3: Read. From the above considerations, this sub-round occurs when the first non Committed instruction is NotReady. Then, the CU submits to memory the read request(s) for all NotReady instructions. Ready packets will simply be discarded. Once all packets are processed, the CU extracts from the memory queue the responses to the read requests submitted during the previous sub-round and sends them to the stations that originated the requests. Then, the instruction templates are shifted down by c stations, dropping the first c instructions, since they have been completely processed. Finally, the state of all active stations is set to Ready and a new round can be started. The time for this step can be bounded as $t_{read} = O(q' + a(M_{exec}) = O(q' + a(M_{exec}))$, where M_{exec} denotes the maximum address accessed during the execution phase.

The time for an entire round can be written as $t_{round} = t_{compute} + t_{write-back} + t_{read} = O(q + a(M_{exec}))$ and the time for the whole execution phase with a total of R rounds is given by

$$T_{exec} = O(R(t_{round})) = O(R[q + a(M_{exec})]).$$

Combining now the bounds for the initialization and the execution phases of the segment, we can bound the time T for the entire first stage as

$$T = T_{init} + T_{exec} = O(R[q + a(M_{max})]),$$

where $M_{max} = \max(M_{init}, M_{exec})$.

Next, we provide some intuition on which properties of a computation are captured by the terms in the right hand side of the above relation.

- *Locality of references* is partially captured by the term M_{max}, which bounds the region of memory accessed during the stage.
- *Branch prediction accuracy* is reflected by q. As the bound shows, values of q larger than r, the number of instructions actually committed, increase the time of the segment without increasing its productivity. As an indirect effect, M_{max} can also increase with q, as later instructions may access higher addresses.
- *Address dependence* impacts the number of rounds R, as argued next. Note that, at the end of the initialization phase, the first instruction has all the correct inputs so that it will execute, commit, and forward a Committed state. Such a state is propagated during that round as long as no instruction detects any new address dependences (thus becoming NotReady). Once an instruction becomes NotReady, no other instruction down-stream is Committed, which will cause another round. A round during which no new address dependences are discovered becomes the last round. Thus, R is one plus the maximum number of address dependences in a chain of (address and functional) dependences[††]. In other words, $R = 1 + D$, where D is the AD-depth of segment, which is in turn at most the AD-depth of the entire instruction stream of the program execution. Observe how two unrelated chains of address dependences are executed concurrently.

[††]However, there is one noteworthy caveat. In the above analysis, we assumed that speculation does not interfere in an adverse manner, in the following sense. Consider an address dependence, $j \xrightarrow{*w} k$. If an intermediate instruc-

Finally, the total execution time for a program can be written as sum of the times of all stages (*i.e.* for a total of S segments)

$$T = \sum_{s=1}^{S} T^s = O(\sum_{s=1}^{S} R^s[q^s + a(M^s_{max})]). \tag{25}$$

In principle, the various terms are uniquely determined once the program, its input (the initial state m_0 of the data memory), and the instruction generation policy (which determines which branches are speculated and the corresponding target, as well as the length q^s of a segment) are known. Analysis might be complex, in general. However, as we will see in the next section, there are interesting algorithms for which the analysis is manageable and leads to rather interesting results.

6 Algorithmic Performance

In this section, we analyze the performance of the SPE for certain classes of algorithms. On the one hand, the analysis will provide us with some appraisal of the potential of the proposed processor. On the other hand, the analysis will yield some insights into the properties of computations that impact upon their execution time.

The SPE can take advantage of both the pipelined nature and the hierarchical nature of the memory to which it is connected. When formulating the performance results for the SPE, we will state the weakest assumptions on memory performance sufficient to obtain the result. Like the SP, the SPE can execute direct-flow programs optimally, as stated next.

Proposition 7 *An SPE with a pipelined memory with latency at most $a(M)$ can execute any* ***direct-flow*** *program of N instructions accessing memory locations with address smaller than M in time $T = O(N + a(M))$, using $n = a(M)$ execution stations.*

We omit the proof, since it is a simple adaptation of the proof of Proposition 6, with the observation that a segment without address dependences and branches is executed by the SPE in one round ($R = 1$). The result of Proposition 7 can naturally be extended to straight-line programs, in terms of their address-dependence depth.

Proposition 8 *An SPE with a pipelined memory with latency at most $a(M)$ and $n = a(M)$ stations can execute any* ***s****traight−line program of N instructions, with address-dependence depth D, accessing memory locations with address smaller than M, in time $T = O(D(N + a(M)))$.*

Again, the argument is a simple adaptation of the arguments in Proposition 6, considering that the number of rounds per stage is $R^s \leq D + 1$. In contrast, one can construct straight-line programs with $D = 1$ whose execution time on the SP processor [11] is $T_{SP} = \Theta(Na(M))$. Thus, the performance of the SPE can be considerably superior to that of the SP.

tion, $I_i, j < i < k$, has an indirect destination operand and falsely speculates that its destination address is w, instruction I_k is forced to discard any speculative contents of location w, even if they are correct. We call this phenomenon *interference*. Since it is presumably unlikely, it makes sense to analyze the behavior of the SPE assuming no interference.

6.1 A Set of Loop Programs

Truly straight-line programs can be useful as conceptual benchmarks, but are of very little significance in practice. There are however a number of interesting algorithms that can be coded naturally as nested loops whose body is essentially straight-line. We introduce a useful class of such programs in the following definition.

Definition 7 *Let $\mathcal{L}$ denote the subset of programs consisting of* ***nested for loops****, whose only branches are those determining whether or not a loop must be exited. Let $\mathcal{L}_D$ be the subset of programs in $\mathcal{L}$ with address dependence depth at most D.*

Remark 1. It is noteworthy to observe that the set $\mathcal{L}_1$ contains a number of key algorithms. For example, a simple analysis would show that $\mathcal{L}_1$ includes standard addition and multiplication of rectangular matrices; bitonic merging and sorting; Cooley-Tukey's FFT; the finite-difference solutions of some important partial differential equations in rectangular domains; and the simulation of digital filters (discrete-time, linear, time-invariant recurrences).

To develop some intuition, let us first consider an $\mathcal{L}_1$ program consisting of just one loop to be iterated k times, thus executing $N = \Theta(k)$ instructions. Let the loop branch be statically predicted to reenter the loop. Then, it is easy to see that an SPE with n stations will break the program into $\lceil N/n \rceil$ segments, each of n instructions; the segment where the loop should be exited will generally include a number of mispredicted instructions. The resulting execution time is $T = O(\lceil N/n \rceil (n + a(M)))$. If $n = a(M)$ and $n = O(N)$, then $T = O(N)$ and performance is optimal to within a constant factor. Instead, if $N = o(n)$, then asymptotically (with respect to the problem size), N becomes much smaller than n. Execution produces only one segment, most of which consisting of mispeculated instructions, resulting in suboptimal execution. By and large, useful programs fall in the optimal case, since the number of instructions typically grows at least with the amount of space used by the program, whence $n = a(M) = O(M) = O(N)$.

Next, let us consider, as an example, the accumulation of the product of two $k \times k$ matrices a and b into a matrix c. Let the program be the straightforward triply nested for-loop, with indices $h, i, j = 0, \ldots, k-1$ and innermost body $c(h,i) = c(h,i) + a(h,j)b(j,i)$. Let us further assume that all loop branches are statically predicted to reenter their respective loops. What is the running time of this simple program on an SPE? We will assume that the matrices and all auxiliary variables are stored in the fastest $M = O(k^2)$ memory locations and we set $n = a(M)$. Then, we analyze how the program is broken up into segments and apply Equation (25). Let ξ be the number of machine instructions executed in each iteration of the innermost loop. (Clearly, ξ is constant w.r.t. k.) The number of segments S can be bounded as follows:

$$S \leq \xi k^3/n + k^2 + k + 1 = O(N/n + k^2). \tag{26}$$

In fact, the term $\xi k^3/n$ is an upper limit to the number of segments where no branch is mispredicted and n useful instructions are executed, considering that the instruction stream has length $N \approx \xi k^3$. The term $k^2 + k + 1$ is the number of times a loop is exited and hence an upper limit to the number of segments with a branch misprediction. Clearly, the size of each segment s is $q^s = n$, with the possible exception of the very last segment ($s = S$), where the halt instruction terminates the segment. Furthermore, we can easily see that the address-dependence depth of the entire program is $D = 1$. Indeed, the references to matrix entries are address-dependent upon the computation of

the array indices, but they do not affect further address computations. Thus, the number of rounds satisfies $R^s \leq 2$, for each segment s. Finally, $M^s_{max} = O(k^2)$ and, since we are not interested in superlinear latency functions, we can assume $a(M^s_{max}) = a(O(k^2)) = O(a(k^2))$. By plugging the relations just derived in Equation (25), we obtain

$$T = O(S(n + a(k^2))) = O(N + Nn/k). \tag{27}$$

We can easily see that, if k grows at least as fast as n (technically, $k = \Omega(n)$), then $T = O(N) = O(k^3)$, which is asymptotically optimal. Otherwise ($k = o(n)$), $T = O(Nn/k)$ grows faster than N, and the result is suboptimal. Using $n = a(M)$ and $M = \Theta(k^2)$, we can see that optimality is achieved when $a(x) = O(\sqrt{x})$, while the suboptimal behavior occurs for larger latency functions, $a(x) = \Omega(\sqrt{x})$. Informally, the suboptimal behavior is due to a number of instructions in the inner loop insufficient to fill the memory pipe; hence latency is not fully hidden.

One solution to the problem just encountered for matrix multiplication consists in transforming the program from a loop nest to a single loop, bringing the number of branch mispredictions down to 1, when the loop is finally exited. This can be accomplished, with the help of predicated instructions, that ensure that the portions of code that are outside the innermost loop are executed only when appropriate. The technique is introduced in the proof of the following proposition and is an extension of *loop coalescing* to loops that are not perfectly nested [33].

Proposition 9 *There exists a* ***predicated coalescing*** *procedure that transforms any program $\mathcal{P} \in \mathcal{L}_D$ into an equivalent program $\hat{\mathcal{P}} \in \mathcal{L}_D$, with just one loop. Furthermore, for any given input, $\hat{N} = O(N)$, where $\hat{N}$ (respectively, N) is the number of instructions executed by $\hat{\mathcal{P}}$ (respectively, $\mathcal{P}$) and $\hat{M} = O(M)$, where $\hat{M}$ (respectively, M) is the number of memory locations accessed by $\hat{\mathcal{P}}$ (respectively, $\mathcal{P}$).*

Proof: We begin by showing how to coalesce two (not necessarily perfectly) nested loops into one loop. In the general case, say with L loops, it is sufficient to repeat the procedure $L - 1$ times, each time coalescing the two innermost loops into one. Thus, let $\mathcal{P}$ be of the following form, where A_1, A_2, and B_1 are suitable code fragments (since $\mathcal{P} \in \mathcal{L}_D$, from Definition 7, the fragments are straight-line; however, coalescing does not rely on this property):

```
for h1 = 0 to H1 − 1 {
   A1
   for h2 = 0 to H2 − 1 {
     A2 }
   B1 }
```

A little reflection will show that the following program $\mathcal{P}'$, where c_1 is a Boolean variable, is indeed equivalent to $\mathcal{P}$.

```
c1 = true; h1 = h2 = 0;
for h = 0 to (H1H2 − 1) {
   if c1 then A1;
   A2;
   h2 = (h2 + 1) mod H2;
   c1 = (h2 = 0);
   if c1 then B1;
   if c1 then h1 = (h1 + 1) mod H1}
```

If, in $\mathcal{P}'$, we replace each if-then construct with an equivalent predicated version, we obtain the desired program $\hat{\mathcal{P}}$, with just one loop with no branch in its body. We can also see that $\hat{N} = O(H_1 H_2) = O(N)$ as well as that $\hat{M} = O(M)$. Finally, we observe that coalescing does not increase the address dependence depth, whence $\hat{\mathcal{P}} \in \mathcal{L}_D$. □

By combining the preceding proposition with the analysis of loop performance given earlier in the section, we obtain the following theorem.

Theorem 3 *A program $\mathcal{P} \in \mathcal{L}_D$ accessing memory locations with addresses smaller than M can be coalesced automatically into an equivalent program $\hat{\mathcal{P}} \in \mathcal{L}_D$, which an SPE with $n = a(M)$ stations executes in time $T = O(D(N + a(M))$, where N is the length of the instruction stream of the original program $\mathcal{P}$ and $a(M)$ is the maximum latency of the pipelined memory.*

As a corollary of Theorem 3, all algorithms mentioned in Remark 1 can be implemented on the SPE, with asymptotically optimal time.

With the notation of Proposition 9, while asymptotically $\hat{N} = O(N)$, the constant of proportionality can be rather large. This constant grows with the ratio of the number of instructions in all loops and the number of those in the innermost loop. In practice, this constant can be reduced, in several ways, all of which can be easily handled by standard compiler techniques. A general approach consists in statically unrolling the innermost loop (which will also eliminate some branches, as a side effect). Another approach that works in some cases consists in coalescing only some of the inner loops, the minimum number sufficient to make the number of iterations in the folded loop grow at least proportionally to $a(M)$. As an example, the reader can verify that coalescing two loops is sufficient to get optimal time for the matrix multiplication program discussed above. When the loop indices have ranges of different size (*e.g.*, multiplication of rectangular matrices), it might be helpful to perform a loop interchange, so that the range size increases from outermost to innermost loop, before performing partial coalescing.

It is interesting to observe how, thanks to the high bandwidth of the pipelined memory, the SPE can execute efficiently the straightforward loop implementation of a number of algorithms, such as matrix multiplication and FFT, which on current machines do require a non trivial exploitation of temporal locality, by blocking or other means, to achieve a good fraction of peak performance. In general, as memory systems provide more bandwidth by supporting larger numbers of simultaneously outstanding references, absolute minimization of the number of misses becomes less crucial, as observed in [43], for matrix multiplication.

6.2 Quicksort

In the algorithms of the previous subsection, one can identify long segments of the instruction stream with no mispredicted branches, so that they can fully hide the worst-case latency of the memory. Clearly, not all algorithms exhibit such a behavior. For example, in recursive algorithms, the number of executed instructions becomes smaller as control goes deeper in the recursion tree, where subproblems of smaller size are being processed. For these algorithms, one can still obtain very good performance if data can be moved to faster regions of memory corresponding to calls where the problem size becomes smaller. Then, access latency becomes smaller and shorter segments are sufficient to hide it. It is also necessary to keep the segment no longer than it needs to be, to avoid severe branch misprediction penalties. To help the IGU to properly tailor the length of the segments that it gives the IEU, we have introduced the semantically neutral segmentsize

instruction, in Section 2.1. This instruction, written as `segmentsize`(ℓ) in the pseudo-code below, provides the IGU with the information that ℓ is a "safe bet" for the length of segments. Then, the IGU simply forms segments of length $q = \min(\ell, n)$, since q cannot exceed the number of stations n. The segmentsize directive is particularly useful when $\ell < n$, as it prevents long segments that contain an early misprediction.

In this subsection, we illustrate the outlined approach, by studying a recursive implementation of quicksort. Even when the number K of items to be sorted is fixed, quicksort exhibits a dynamic nature, with the number of executed comparisons varying with input data, from an average $\theta(K \log K)$ to a worst case $\theta(K^2)$. It can be shown that the address dependence depth D of the algorithm is proportional to the depth of the recursion tree, thus ranging between $\theta(\log K)$ and $\theta(K)$. Nevertheless, the SPE can hide the memory latency and achieve running time proportional to the number of comparisons (hence, of instructions), for any given input. On average, execution time will then be $\theta(K \log K)$, as long as $a(x) = O(x)$. When contrasted with the $\Omega(K \log^2 K)$ time lower bound established by [3] for sorting in the Block Transfer model with $a(x) = O(x)$, our result provides further evidence of the advantage of arbitrary pipelined transfer over block transfer.

Theorem 4 *The SPE, with memory access function $a(x) = O(x)$ and $n = a(K)$ stations, can* ***quicksort*** *an array of K elements placed in the first $O(K)$ memory locations in time $T_{qsort} = O(C_{qsort})$, where C_{qsort} is the number of comparisons executed by the quicksort algorithm.*

Proof: For simplicity, we describe and analyze the algorithm assuming that the number, n, of the SPE stations is proportional to K rather than to $a(K)$. The adaptation to the latter case is straightforward and is simply based on naturally breaking up instructions segments longer than n.

Data layout. We will make use of a constant number, s, of arrays, each of K one-word components, with the understanding that their memory layouts are interleaved. Thus, if X is the j-th such array, then for $i = 1, 2, \ldots, K$, $X[i]$ is mapped into memory location $j + s(i-1)$. This layout guarantees that the latency for accessing $X[i]$ is $a(j + s(i-1)) = O(i)$.

Recursive formulation. We define a procedure `pivot`(k) which uses $A[1]$ as the pivot and reorganizes $A[1], \ldots, A[k]$ so that all the elements smaller than the pivot come before all elements greater than the pivot. We define a procedure `swap`(k_1, k_2) which cyclically shifts $A[1], \ldots, A[k_1 + k_2]$ by an amount k_2 ($A[1]$ goes to $A[1 + k_2]$).

We define a procedure `qsort`(k) which will sort $A[1], \ldots, A[k]$, in nondecreasing order:

```
qsort(k)
  if k ≤ 1 then return;
  segmentsize(k);
  pivot(k);
  segmentsize(k1);
  qsort(k1);
  segmentsize(k);
  swap(k1, k2);
  segmentsize(k2);
  qsort(k2);
  segmentsize(k);
  swap(k2, k1);
end
```

Here, k_1 denotes the number of elements smaller than or equal to the pivot and $k_2 = k - k_1$ denotes the number of elements larger than the pivot. The segmentsize instruction indicates that the next portion

of the instruction stream can be profitably split into segments of length proportional to the amount of data being processed. Indeed, from the implementation of the various procedures, it becomes clear that they consist of loops that iterate over all data. The segmentsize instructions simply take constant time. Thus, from the above recursive definition of quicksort one can easily obtain a recurrence relation for its running time:

$$T_{qsort}(k) = T_{qsort}(k_1) + T_{qsort}(k_2) + T_{pivot}(k) \\ + T_{swap}(k_1, k_2) + T_{swap}(k_2, k_1) + T_{overhead}(k).$$

By $T_{overhead}(k)$ we denote any time not accounted for by the remaining terms, essentially the overhead to manage the recursion. Our goal is to show to bound the last four terms in the above recurrence by $O(k)$, to obtain:

$$T_{qsort}(k) = T_{qsort}(k_1) + T_{qsort}(k_2) + O(k). \tag{28}$$

Since, as is well known, the number C_{qsort} of comparisons performed by quicksort also satisfies a recurrence relation of the same type, it is straightforward to conclude that $T_{qsort} = O(C_{qsort})$, as claimed by the theorem statement.

Pivot. To implement `pivot`(k) we use an auxiliary array B, where the data are first written as they are partitioned with respect to the pivot $A[1]$; then, they are copied back to A. This approach makes it easier to exploit concurrency than when performing the operation in place. A code fragment implementing the partition is as follows:

$low = 1$;
$high = k$;
`for` $j = k$ `downto` 1 {
  $c = (A[j] \leq A[1])$;
  `if` c `then` $B[low] = A[j]$;
  `if` $(1 - c)$ `then` $B[high] = A[j]$;
  $low = low + c$;
  $high = high - (1 - c)$;}

The if statements can be easily translated into predicated write instructions, whereby avoiding actual branches. A little analysis will show that the instruction segment resulting when executing the for loop has length $\theta(k)$ and has constant address-dependence depth. Copying back $B[1...k]$ into $A[1...k]$ can be done in a similar way. In conclusion, `pivot`(k) is accomplished by executing segments of $q = O(k)$ instructions, involving addresses no greater than $M = O(k)$, with constant AD-depth, whence $T_{pivot}(k) = O(k)$. (It might be interesting to observe that, in the global instruction stream generated by quicksort, there are many address dependences, typically between instruction of one call and instructions of its children calls. However, these dependences span across segments and are accounted for as part of the initialization of the segments.)

Swap. Following an approach analogous to the one used for `pivot`(k), one can show that $T_{swap}(k_1 + k_2) = O(k_1 + k_2)$.

Recursion stack. Some care has to be exercised in order to manage the recursion stack within the desired time bounds. For example, if insertions cause the address of the top of the stack to increase, then the calls handling small subproblems end up suffering a higher stack overhead, resulting in unacceptable performance. On the other hand, if we let the stack grow toward decreasing addresses, since quicksort might generate $O(K)$ simultaneous entries in the recursion stack, the bottom of the stack has to be placed at an $O(K)$ address, which also results in unacceptable performance. One solution is to implement the stack as a doubly linked list to be stored within a suitable number of arrays (the exact number depending on the amount of information needed per stack frame). When a call of size k creates the stack entry for the child call of size k_1, this entry is placed at the k_1-th component of the stack arrays. Detailed implementation and analysis show that $T_{overhead}(k) = O(k)$ is attained, essentially resulting from a

constant number of accesses performed at addresses $O(k)$ (no concurrency between these accesses is assumed). □

6.3 Merging

In this subsection we focus on the implementation of the well-known algorithm for merging two sorted sequences. While very simple, this algorithm is rather instructive in our context, since it has a large address-dependence depth, proportional to the number of executed instructions. We will see how a judicious use of (software) prefetching, which exploits both the hierarchical and the pipelined nature of the memory, can yield an implementation with a low CPI. However, whether this basic merging algorithm can be implemented with constant CPI on the SPE remains an open question.

We assume that the input sequences are monotone non-decreasing and are initially stored in two arrays $A[0 \ldots K_A - 1]$ and $B[0 \ldots K_B - 1]$. The output is to be written in an array $C[0 \ldots K - 1]$, where $K = K_A + K_B$. We will assume that suitable "sentinel" values will be appended at the end of each input sequence, *i.e.,* $A[K_A] = B[K_B] = +\infty$. For the data layout, we follow the same approach of interleaving used for quicksort (see proof of Theorem 4). Then, merging can be accomplished by the following code:

1. $i_A = i_B = i_C = 0$;
2. `for` $i_C = 0$ `to` $K_A + K_B - 1$ {
3. $c = (A[i_A] \leq B[i_B])$;
4. `if` c `then` $C[i_C] = A[i_A]$;
5. `if` $(1 - c)$ `then` $C[i_C] = B[i_B]$;
6. $i_A = i_A + c$;
7. $i_B = i_B + (1 - c)$;}

The if statements 4 and 5 can be easily translated into predicated write instructions, whereby avoiding actual branches. We observe that statement 3 exhibits (loop-carried) address dependences upon statements 6 and 7, via the variables i_A and i_B. Thus, the AD-depth is $D = O(K)$. The instruction stream can be partitioned into $S = O(K/q)$ segments of size q, each with AD-depth $O(q)$, thus requiring $R = O(q)$ rounds on the SPE. Clearly, the amount of space used by the algorithm is bounded as $M_{max} = O(K)$. The program exhibits negligible temporal locality: in fact, for at least a constant fraction of the segments, a constant fraction of the accesses will target addresses proportional to K. From simple manipulations of Equation (25), the execution time is then

$$T = O(SR(q + a(M_{max})) = O(K(q + a(K))),$$

which is clearly minimized to $T = O(K * a(K))$, when $q = 1$, that is, when the SPE behaves essentially as the canonical processor.

Intuitively, the address dependences prevent any significant overlap of the memory references that access the operands for the comparisons. However, it is possible to reduce the latency of such references, by suitably moving the data to a region of the memory with faster access, before they are needed for the comparisons. The key property of the merging algorithm that is helpful in this context is the partial predictability of its pattern of data usage. Specifically, with reference to the above program, after any iteration of the for-loop, we know that the input elements read

in the next h iterations belong to the subarrays $A[i_A \ldots i_A + h - 1]$ and $B[i_B \ldots i_B + h - 1]$. By prefetching these subarrays into $O(h)$ faster memory locations, the latency of subsequent accesses on account of the comparisons is reduced from $a(O(K))$ to $a(O(h))$. The performance advantage arises from the fact that prefetching essentially consists of a set of copy instructions with no dependences among themselves, which can therefore be fully pipelined. In summary, the output array C can be computed in K/h phases, each prefetching h elements from A and h elements from B and performing h comparisons to obtain h elements of C. The prefetching idea can be applied recursively, until sequences of constant length are obtained. Some care is required to specify all the details, particularly to determine the layout of the data at each level of the recursion. We omit these details here, since they would not contribute new insights. (A systematic treatment of memory management in the context of pipelined hierarchy can be found in [9].)

The running time for the version of merging with recursive prefetching outlined above satisfies the following recurrence relation:

$$T_{merge}(K) = O((K/h)[a(K) + h + T_{merge}(h)]). \tag{29}$$

This equation reflects the fact that, for each of K/h phases, there is a prefetching activity involving $O(h)$ pipelined accesses to data placed at addresses no greater than $O(K)$ and a recursive call with output size h. To complete the specification of the outlined implementation of merging, we choose the value of the free parameter h as $h = a(K)$. Informally, we are balancing the "depth" $a(K)$ of the memory pipe and the number, h, of pipelined accesses. Then, Equation 29 becomes

$$T_{merge}(K) = O(K + K * T_{merge}(a(K))/a(K)). \tag{30}$$

The latter recurrence can be solved by standard techniques for many specific shapes of the memory access function $a(x)$. The result for a few cases of interest is reported in the next theorem.

Theorem 5 *The SPE, with memory access function $a(x) = O(x)$ and $n = a(K)$ stations, can* ***merge*** *two sorted sequences with a total of K elements with the following performance:*

1. *If $a(x) = O(x)$, then $T_{merge} = O(K \log K / \log\log K)$.*
2. *If $a(x) = O(x^\alpha)$, with $0 < \alpha < 1$, then $T_{merge} = O(K \log\log K)$.*
3. *If $a(x) = O(\log x)$, then $T_{merge} = O(K \log^* K)$.*

It is natural to wonder whether a more sophisticated implementation of the merging algorithm on the SPE can achieve $T_{merge} = O(K)$. Otherwise, it would be interesting to establish a superlinear lower bound. This objective, however, does not appear to be straightforward: to the best of our knowledge, no lower bound techniques have been previously considered that could be easily adapted to capture the constraints arising from address dependence.

7 Conclusions

As technological evolution leads to machines increasingly constrained by signal speed, the RAM model becomes increasingly inadequate, even in the context of serial computation. The consequences are twofold. On the one side, computer architectures become more complex, to exploit concurrency and locality, in order to approximate the constant rate of instruction execution of the

ideal RAM. On the other side, algorithm design has to be based on new models of computation that reflect the new architectures, to expose concurrency and locality. In principle, machines and algorithms ought to be designed jointly, to achieve optimal performance under the relevant constraints. In this paper, we have developed a framework bridging machine and algorithm design. In this framework, algorithmic performance has been analyzed for the proposed processor and memory organizations, in terms of the problem input size and of the corresponding machine size.

The focus on general-purpose machines poses a methodological question: Which algorithms or programs should be chosen as the performance benchmarks? Rather than preselecting a set of benchmarks, we have taken the approach to measure progress by the increase of the class of programs that the designed machine can execute in time proportional to that of the ideal RAM. This approach has lead to the successive consideration of direct-flow programs, of programs with constant address-dependence depth, and of programs whose execution can be partitioned into segments of constant address-dependence depth and of length at least proportional to the maximum latency of any memory access within the segment. These classes can be dealt with very efficiently by the proposed SPE machine. However, there are clearly algorithms whose running time on the SPE is superlinear in the number of executed instructions. Intuitively, superlinearity can arise from suitable combinations of long address-dependence chains, a high frequency of poorly predicatable branches, and a small degree of locality of references. It remains an open challenge to capture this intuition into a technique to derive lower bounds.

Further algorithmic studies on the SPE could clearly shed further light on the interplay of address dependence, branches, and temporal locality. However, ample room remains to search for improved processors as well. For example, there are programs that run faster on the SP than they run on the SPE and vice versa. Qualitatively speaking, the SP will be more efficient for segments where address dependence chains contain many instructions indirectly accessing fast memory. In contrast, the SPE tends to be more efficient in the presence of many chains, each with few instructions indirectly accessing slow memory. In the latter case, speculative evaluation enables overlapping memory latencies incurred by different chains. In the former case, there is little memory concurrency to exploit and the speculative evaluation overheads are incurred with no performance advantage. It is then plausible that a judicious combination of the ideas behind the SP and the SPE processors could lead to solutions that are better than both.

More generally, our work is far from exhausting the possibilities afforded by physically realizable machines. In fact, in order to begin the study of the relevant issues in a relatively simple setting, we have restricted our attention to machines that exhibit artificial bottlenecks, whose removal should definitely be explored in further work. Specifically, we observe that while memory pipelining increases the bandwidth of what Backus [8] has dubbed the von Neumann tube, more aggressive steps can be taken in this direction. Indeed, in d-dimensional machines exhibiting a von Neumann tube (*i.e.,* where processor and memory lie on opposite sides of a suitable geometric cut), the bandwidth of the tube could grow proportionally to $M^{(1-d)/d}$, rather than being constant, as in our designs, or asymptotically vanishing, as in non-pipelined designs. A more radical step eliminates the bottleneck altogether, with machine organizations that interleave memory and processing regions throughout the layout. Ultimately, it is natural to extend the entire approach to machines that execute explicitly parallel programs, combining the insights obtained during the quest for the best physical realization of the sequential RAM with decades of results in the area of parallel computing.

Acknowledgements. The work of G. Bilardi has been supported in part by Project 2006092119

MAINSTREAM of MIUR, by FP6-IST/IP Project 15964 *AEOLUS* of the European Union, and by the IBM Research Division.

The authors would like to thank Kieran Herley, Keshav Pingali, and Geppino Pucci for insightful discussions.

References

[1] F. Abolhassan, R. Drefenstedt, J. Keller, W.J. Paul, and D. Scheerer. On the Physical Design of PRAMs. *Computer Journal*, 36(8), pp. 756-762, December 1993.

[2] A. Aggarwal, B. Alpern, A.K. Chandra and M. Snir. A Model for Hierarchical Memory. In *Proc. of the 19th ACM Symposium on Theory of Computing*, (1987), 305–314.

[3] A. Aggarwal, A.K. Chandra and M. Snir. Hierarchical Memory with Block Transfer. In *Proc. 28th Annual Symposium on Foundations of Computer Science*, (1987), 204–216.

[4] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kauffman, 2002.

[5] B. Alpern, L. Carter, E. Feig and T. Selker. The Uniform Memory Hierarchy Model of Computation. In *Algorithmica*, vol. 12, (1994), 72-129.

[6] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, B. Smith. The Tera Computer System. ACM International Conference on Supercomputing,pp. 1-6, June 1990.

[7] N.M. Amato, J. Perdue, A. Pietracaprina, G. Pucci, and M. Mathis. . Predicting performance on SMP's. A case study: The SGI Power Challenge. *In Proc. International Parallel and Distributed Processing Symposium, IPDPS 2000*, Cancun, MEX, pages 729-737, May 2000.

[8] J. Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs (1977 ACM Turing Award Lecture). *Communications of the ACM*, 21, 8 (Aug 1978), 613-641.

[9] G. Bilardi, K. Ekanadham, P. Pattnaik. On the Computational Power of Pipelined Hierarchical Memories. In *Proc. of the 13th ACM Symposium on Parallel Algorithms and Architectures*, Crete, Greece (2001).

[10] G. Bilardi, K. Ekanadham, P. Pattnaik. Optimal Organizations for Pipelined Hierarchical Memories. In *Proc. of the 14th ACM Symposium on Parallel Algorithms and Architectures*, Winnipeg, Canada (2002).

[11] G. Bilardi, K. Ekanadham, P. Pattnaik. An Address Dependence Model of Computation for Hierarchical Memories with Pipelined Transfer. In *Proc. of the 19th IEEE International Parallel and Distributed Processing Symposium*, Workshop 8, Volume 09 (2005).

[12] G. Bilardi, K. Ekanadham, P. Pattnaik. The Speculative Prefetcher and Evaluator Processor for Pipelined Memory Hierarchies. *IWIA 2006, International Workshop on Innovative Architectures for Future Generation High-Performance Processors and Systems*, The Big Island, Hawaii, January 2006.

[13] G. Bilardi, E. Peserico. An Approach toward an Analytical Characterization of Locality and its Portability. *IWIA 2000, International Workshop on Innovative Architectures*, Maui, Hawaii, January 2001.

[14] G. Bilardi, E. Peserico. A Characterization of Temporal Locality and its Portability Across Memory Hierarchies. *ICALP 2001, International Colloquium on Automata, Languages, and Programming*, Crete, July 2001.

[15] G. Bilardi, A. Pietracaprina, and P. D'Alberto. On the Space and Access Complexity of Computation Dags. *26th Workshop on Graph-Theoretic Concepts in Computer Science*, Konstanz, Germany, June 2000.

[16] G. Bilardi and F.P. Preparata. Horizons of Parallel Computation. *Journal on Parallel and Distributed Computing*, Vol. 27, 172-182, 1995.

[17] G.Bilardi and F.Preparata. Processor-Time Tradeoffs under Bounded-Speed Message Propagation. Part I: Upper Bounds. Theory of Computing Systems, Vol. 30, 523-546, 1997.

[18] G.Bilardi and F.Preparata. Processor-Time Tradeoffs under Bounded-Speed Message Propagation. Part I: Lower Bounds. Theory of Computing Systems, Vol. 32, 531-559, 1999.

[19] B. Chazelle and L. Monier. A Model of Computation for VLSI with Related Complexity Results. *Journal of the ACM*, 32(3):573-588, July 1985.

[20] S.A. Cook and R.A. Reckhow. Time Bounded Random Access Machines. *Journal of Computer System Science*, 7:354-375, 1973.

[21] J. Fotheringham. Dynamic storage allocation in the ATLAS computer, including an automatic use of a backing store. *Communication of the ACM*, 4, 10, pp. 435-436, (1961).

[22] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-Oblivious Algorithms. *Proc. 40th Annual Symposium on Foundations of Computer Science*, (1999).

[23] D.S. Henry, B.C. Kuszmaul, and V. Viswanath. The Ultrascalar Processor: An Asymptotically Scalable Superscalar Microarchitecture. *Proceedings ARVLSI'99*, Atlanta GA, March 21-24, pp. 256-273, IEEE Computer Society, 1999.

[24] J.W. Hong and H.T. Kung. I/O Complexity: the Red-Blue Pebbling Game. In *Proc. of the 13th ACM Symposium on Theory of Computing*, (1981), 326–333.

[25] D.J. Kuck. *The Structure of Computers and Computations.* Volume 1. Wiley, 1978.

[26] B.C. Kuszmaul, D.S. Henry, and G. H. Loh. A Comparison of Asymptotically Scalable Superscalar Processors. *Theory of Computing Systems*, 35:129-150, 2002.

[27] C. Leiserson. Area-Efficient VLSI Computation. *Ph.D. Thesis*, Department of Computer Science, Carnegie Mellon University, October 1981.

[28] F. Luccio and L. Pagli. A model of sequential computation with pipelined access to memory. *Mathematical Systems Theory*, 26(4), 343–356, 1993.

[29] R.L. Mattson, J.Gecsei, D.R. Slutz, and I.L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM System Journal*, No. 2, 78–117, 1970.

[30] C. Metcalf. Data prefetching: a cost/performance analysis. *Area Exam*, MIT, October 1993.

[31] V. Milutinovic and M. Valero (Guest Eds.). Special Issue on Cache Memory and Related Problems. *IEEE Transactions on Computers*, February 1999.

[32] J. von Neumann. First Draft of a Report on the EDVAC. *Moore School of Electrical Engineering, University of Pennsylvania*, June 1945.

[33] C. Polychronopoulos. Loop Coalescing: a Compiler Transformation for Parallel Machines. *Proceedings of the 1987 International Conference on Parallel Processing*, St. Charles, IL, August 1987.

[34] S.S. Przybylski. *Cache and Memory Hierarchy Design. A Performance Directed Approach.* Morgan Kaufmann Publishers, Inc. Palo Alto, CA 1990.

[35] J.E. Savage. *Models of Computation. Exploring the Power of Computing* Addison-Wesley, Reading, MA, 1998.

[36] B.J. Smith. Architecture and applications of the HEP multiprocessor system. *Signal Processing* IV, 298 (1981), 241-248.

[37] L.G. Valiant. General purpose parallel architectures, *Handbook for Theor. Computer Science*, (J.van Leeuwen, ed.), Elsevier-MIT Press, 1990.

[38] J.S. Vitter. External Memory Algorithms. Invited paper in *Proc. 6th Annual European Symposium on Algorithms*, (G.Bilardi et al. Eds.), Springer Verlag, Venice, August 1998, 1–25.

[39] R.C.Whaley and J.J.Dongarra. Automatically Tuned Linear Algebra Software. http://www.netlib.org/atlas/index.html

[40] M. Wilkes. The Memory Gap. *Workshop on Solving the Memory Wall Problem, held in conjunction with ISCA 2000*, June 2000.

[41] M.Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1995.

[42] W.Wulf and S.McKee. Hitting the Memory wall: Implications of the Obvious. *Computer Architecture news*, 23(1):20-24, March 1995.

[43] K. Yotov, T. Roeder, K. Pingali, J. Gunnels, F. Gustavson. An experimental comparison of cache-oblivious and cache-conscious programs. In *Proc. of the 19th ACM Symposium on Parallel Algorithms and Architectures*, San Diego, California (2007).

[44] C.B.Zilles and G. Sohi. A programmable Co-processor for Profiling. *International Symposium on High-Performance Computer Architecture* (HPCA-7), Jan 2001.

[45] C.B.Zilles and G. Sohi. Master/Slave Speculative Parallelization. *35th International Symposium on Micro Architecture* (MICRO-35), Nov 2002.